VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Cenová optimalizace zpracování XML dotazů v Radegast XDB

Cost-based Optimization of XML Query Processing in Radegast XDB

Pavel Ptáček

Diplomová práce

Vedoucí práce: Ing. Petr Lukáš, Ph.D.

Ostrava, 2021

Abstrakt

Tato diplomová práce se zabývá cenovou optimalizací algoritmů pro sestavení plánu vykonání XML dotazu. Práce se zabývá popisem algoritmů binárních spojení a také popisem algoritmů pro sestavení plánu XML dotazu. Tyto algoritmy jsou následně rozšířeny o cenovou a heuristickou optimalizaci. Součástí práce je experimentální testování vylepšení uvedených algoritmů.

Klíčová slova

XML, XPath, XQuery, zpracování XML dotazů, binární spojení, plán vykonání XML dotazu, cenová optimalizace

Abstract

This diploma thesis deals with cost-based optimization of algorithms for building of XML query plan. Thesis deals with the description of binary join algorithms and also with the description of algorithms for building of XML query plan. These algorithms are then extended by cost-based and heuristic-based optimizations. Part of the work is experimental testing of improvements to these algorithms.

Keywords

XML, XPath, XQuery, XML query processing, binary join, XML query plan, cost-based optimization

Poděkování

Rád bych na tomto místě poděkoval mému vedoucímu práce Ing. Petru Lukášovi, Ph.D. za rady a velmi užitečnou pomoc při řešení problémů.

Obsah

Seznam použitých zkratek a symbolů

AD	–	vztah předek – potomek mezi XML uzly (ancestor – descendant)
CPU	–	Central Processing Unit
FP	–	plně zřetězené plány (fully-pipelined)
PC	–	vztah rodič – přímý potomek mezi XML uzly (parent – child)
RAM	–	Random Access Memory
SOAP	–	Simple Object Access Protocol
SGML	–	Standard Generalized Markup Language
TPQ	–	Twig Pattern Query
XML	–	Extensible Markup Language
XHTML	–	Extensible Hypertext Markup Language

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Cílem této diplomové práce je implementace vybraných algoritmů pro sestavení plánu dotazu. Algoritmy budou vytvářet optimalizované plány vykonání dotazu s využitím statistik XML kolekce. Algoritmy budou implementovány do nativní XML databáze Radegast XDB.

První část této práce je věnována vysvětlení základních pojmů, které se v práci vyskytují. Je vysvětlen značkovací jazyk XML, dotazovací jazyky XPath a XQuery a další související informace. XML dotazy se obvykle modelují pomocí Twig Pattern Query, který je v této části rovněž vysvětlen.

Jedním z přístupů ke zpracování XML dotazů pomocí TPQ je přístup založený na binárních spojeních, který rozkládá TPQ do několika algoritmů. Tyto algoritmy, nazývané binární spojení, jsou popsány ve druhé části této práce. Celkově jsou v této části představeny tři druhy algoritmů binárních spojení.

Algoritmy binárních spojení popsané v druhé části samy o sobě neumí zpracovat TPQ dotaz. K vyhodnocení TPQ je nutné sestavit plán vykonání dotazu, který se skládá z vhodně uspořádaných algoritmů binárních spojení. Ve třetí části této práce je představeno několik algoritmů, které slouží k sestavení zmíněného plánu vykonání dotazu. Nejprve jsou představeny algoritmy bez cenové optimalizace. Tyto algoritmy jsou poté rozšířeny o cenovou a heuristickou optimalizaci, které vedou k efektivnějšímu zpracování dotazů. Všechny algoritmy jsou podrobně popsány pomocí pseudokódu.

V poslední části práce jsou algoritmy prezentované v předchozí části otestovány a porovnány pomocí experimentálních dotazů. Algoritmy s cenovou a heuristickou optimalizací jsou porovnány na základě jejich zrychlení oproti původním neoptimalizovaným algoritmům. Dále je také změřen a porovnán poměr času stráveného optimalizací k celkovému času vykonání dotazu. Toto porovnání ukáže, zda není optimalizace náročnější než samotné vykonání dotazu, což by nebylo žádoucí. V závěru této části jsou všechny implementované algoritmy souhrnně porovnány a ohodnoceny dle jejich výkonnosti.

Kapitola 2

Základní pojmy

V této kapitole jsou popsány základní pojmy, které se vyskytují v této práci.

2.1 Extensible Markup Language

Extensible Markup Language, zkráceně XML, česky rozšiřitelný značkovací jazyk, je formální jazyk, který definuje pravidla pro kódování dokumentů ve formátu, který je lehce čitelný jak pro člověka, tak pro počítač. Návrhové cíle XML kladou důraz na jednoduchost, obecnost a použitelnost.

Tento jazyk byl vyvinut a standardizován konsorciem W3C v roce 1998 [1] jako následovník jazyka SGML, který byl pro koncového uživatele nepřehledný, některé jeho vlastnosti nebyly užitečné a implementace parseru byla příliš složitá. XML byl oproti SGML zjednodušen a byly mu přidány některé užitečné vlastnosti.

XML se používá především pro serializaci dat a snadnou výměnu dat mezi aplikacemi přes Internet. Používá se například v XHTML, SOAP, Office Open XML, Google Maps. Příklad XML dokumentu je na obrázku 2.1. XML dokument se díky své stromové struktuře dá znázornit pomocí stromu. Příklad XML stromu se nachází na obrázku 2.2. Názvy uzlů jsou pro přehlednost doplněny o indexy, tyto indexy ale součástí uzlů v XML dokumentu nejsou. Pod názvy uzlů se nachází čísla v hranatých závorkách. Jedná se o značky uzlu, které budou podrobněji popsány v kapitole 2.6.

2.1.1 Terminologie XML

Nejpoužívanější pojmy z oblasti XML jsou následující:

- **Uzel** – uzel ve stromové struktuře XML dokumentu.
- **Tag** – označuje začátek nebo konec uzlu. Existují tři druhy tagů:
  - počáteční tag, který označuje začátek uzlu, např. `<email>`,
  - koncový tag, který označuje konec uzlu, např. `</email>`,

```xml
<emails>
   <email id="1">
      <from>Chandler</from>
      <to>Michaela</to>
      <subject>Předmět prvního emailu</subject>
      <body>Obsah prvního emailu</body>
   </email>
   <email id="2">
      <from>Jack</from>
      <to>Monica</to>
      <subject>Předmět druhého emailu s~přílohou</subject>
      <body>Obsah druhého emailu s~přílohou</body>
      <attachments>
         <attachment>
            <filename>soubor.txt</filename>
         </attachment>
      </attachments>
   </email>
   <email id="3">
      <from>Claire</from>
      <to>David</to>
      <subject>Předmět třetího emailu</subject>
      <body>Obsah třetího emailu</body>
   </email>
</emails>
```

Obrázek 2.1: Příklad XML dokumentu

  - prázdný tag, který se používá tehdy, pokud uzel neobsahuje žádné další uzly, např. `<email />`.

- **Sousední uzel** – sousedními uzly určitého uzlu jsou všichni jeho přímí potomci a jeho rodič.

- **Atribut** – atributy jsou součástí uzlu, které se zapisují do počátečního nebo prázdného tagu a specifikují vlastnosti uzlu, např. atribut `id` u uzlu email – `<email id="1">`.

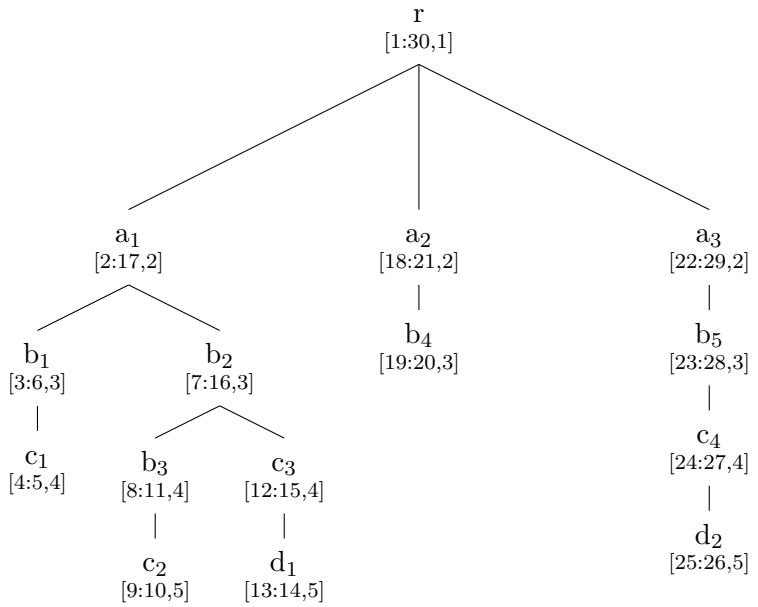
2.2 XPath

XPath je dotazovací jazyk pro získávání dat z XML dokumentu. Tento jazyk byl definován a standardizován konsorciem W3C v roce 1999 [2]. XPath dotaz se skládá z výrazů popisujících cestu k dotazovaným XML uzlům. Tato cesta je zapisována jako posloupnost kroků mezi jednotlivými uzly. Krok je zapisován výrazem `osa::test[predikát]`. `osa` specifikuje směr pohybu mezi uzly a může nabývat jedné z hodnot uvedených v tabulce 2.1. Po ose následuje `test`, oddělený dvěma dvojtečkami, který může obsahovat název uzlu nebo nějaký obecný výraz, např. znak `*` pro výběr jakéhokoliv uzlu, příkaz `comment()` pro získání XML komentáře aj. Poté následuje `[predikát]`, který specifikuje podmínku, kterou musí daný uzel splňovat. Predikát se může skládat z jedné nebo více podmínek. Každý krok se vyhodnocuje vzhledem ke kontextovým uzlům, které odpovídají množině uzlů získané z výstupu předchozího kroku.

Na začátku vyhodnocování XPath výrazu je na vstupu jeden uzel, který reprezentuje celý XML dokument. Na tomto uzlu se provede první krok. Dle osy, specifikované parametrem `osa`, se ze vstupu vybere množina uzlů, pro které platí strukturální vztah mezi těmito uzly a kontextovými uzly. Z této množiny uzlů se vyberou uzly, které odpovídají výrazu, který je specifikovaný parametrem `test`. Následně se na výsledné množině provede selekce dle podmínek uvedených v parametru `[predikát]`. Tato množina je výsledkem aktuálního kroku a stává se vstupem do dalšího kroku, ve kterém se postup opakuje.

2.3 XQuery

XQuery (XML Query), je dotazovací funkcionální programovací jazyk, který provádí dotazy a transformuje data nad XML dokumentem. Tento jazyk byl vyvinut a standardizován konsorciem W3C v roce 2007 [3]. XQuery je nadmnožinou XPath, což znamená, že jakýkoliv XPath dotaz je validním XQuery dotazem. Na rozdíl od jazyka XPath, který pouze filtruje vstupní množinu uzlů na nějakou její podmnožinu, jazyk XQuery navíc umožňuje různé transformace výstupu, které zahrnují vytváření nových uzlů nebo úpravy stávajících. Těmito úpravami je ale možné měnit pouze výstup XQuery dotazu, provádět úpravy samotného XML dokumentu pomocí XQuery možné není. K tomuto slouží XQuery Update Facility.



Obrázek 2.2: Příklad XML stromu

Název osy	**Význam**	**Zkratka**
ancestor	předek	
ancestor-or-self	předek nebo aktuální uzel	
attribute	atribut	@
child	přímý potomek	
descendant	potomek	
descendant-or-self	potomek nebo aktuální uzel	//
following	následující uzel	
following-sibling	následující sourozenec	
namespace	jmenný prostor	
parent	přímý předek	..
preceding	předcházející uzel	
preceding-sibling	předcházející sourozenec	
self	aktuální uzel	.

Tabulka 2.1: XPath osy

```
for $x in /r/a
let $y := $x/b[./c and .//d]
where $y > 1
order by $x
return ($x, $y);
```

Obrázek 2.3: Příklad XQuery dotazu

V XQuery se kromě XPath výrazů používají FLWOR výrazy. Název FLWOR vychází z prvních písmen následujících klauzulí:

- **for** – procházení vstupních uzlů
- **let** – přiřazení sekvence uzlů do proměnné
- **where** – filtrace uzlů na základě pravdivostního výrazu
- **order by** – seřazení uzlů
- **group by** – seskupení uzlů do skupin (dostupné od XQuery verze 3.0)
- **return** – výstup

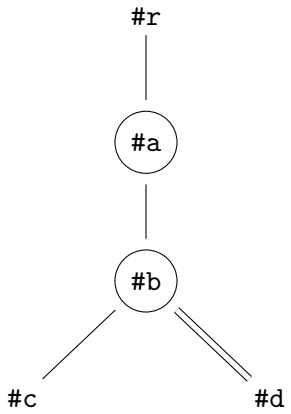
Příklad XQuery dotazu je na obrázku 2.3. Dotaz se vztahuje ke XML dokumentu, který je znázorněn pomocí XML stromu a nachází se na obrázku 2.2. Pomocí klauzule `for` se začnou procházet uzly specifikované XPath výrazem `/r/a`. Sekvence těchto uzlů se nachází v proměnné `$x`. Následně se pomocí klauzule `let` vytvoří proměnná s názvem `$y`, do které se vloží sekvence uzlů specifikovaná XPath výrazem `$x/b[./c and .//d]`. V tomto XPath výrazu je použita proměnná `$x`. Dále je použita klauzule `where`, která specifikuje, že se mají vybrat pouze ty uzly, pro které platí, že hodnota proměnné `$y` je větší než 1. Po této podmínce následuje klauzule `order by`, pomocí které se výsledek seřadí podle uzlů v sekvenci `$x`. Posledním krokem je klauzule `return`, která z uzlů v sekvencích `$x` a `$y` vytvoří dvojice a tyto dvojice vloží na výstup. Uvedený postup pouze demonstruje, jaký má být výsledek XQuery dotazu. Samotný algoritmus zpracování takového dotazu totiž může pracovat úplně jinak.

2.4 Twig Pattern Query

Twig Pattern Query, zkráceně TPQ, je zjednodušený model XQuery dotazu. Jedná se o strom $Q = (V_Q, E_Q)$, kde V_Q je množina dotazovacích uzlů a E_Q je množina dotazovacích hran. Jednotlivé uzly mají specifikován vyhledávací predikát a také informaci, zda se jedná o výstupní nebo nevýstupní uzel. U dotazovacích hran je zaznamenána informace o vztahu mezi uzly – *PC* nebo *AD*. Tyto vztahy mezi uzly jsou popsány v kapitole 2.5.

Příklad TPQ se nachází na obrázku 2.4. Tento TPQ odpovídá XQuery dotazu na obrázku 2.3 a dotazuje XML dokument na obrázku 2.2. Jednoduchá hrana mezi uzly značí PC vztah, dvojitá hrana značí AD vztah. Pokud je uzel zakroužkovaný, jedná se o výstupní uzel. Znak `#` před názvem uzlu značí, že se jedná o test názvu uzlu. Výsledek TPQ je zobrazen v tabulce 2.2.

Pro formální definici výsledku TPQ je zapotřebí nadefinovat dva nové pojmy – *n-tice úplné shody* a *výstupní n-tice*. *N-tice úplné shody* mezi TPQ a XML dokumentem je taková n-tice datových uzlů, pro kterou platí, že jednotlivé dotazovací uzly jsou mapovány na datové uzly s příslušnými názvy a zároveň strukturální vztahy mezi dotazovacími uzly odpovídají strukturálním vztahům mezi datovými uzly. *Výstupní n-tice* vznikne z n-tice úplné shody vynecháním datových uzlů, které neodpovídají výstupním uzlům dotazu. Jedná se tedy o projekci n-tice úplné shody. *Výsledek* TPQ je potom množina všech výstupních n-tic.



Obrázek 2.4: Příklad TPQ

#x	#y
a_1	b_2
a_3	b_5

Tabulka 2.2: Výsledek TPQ

2.5 Vztahy mezi XML uzly

Jednotlivé uzly v XML stromu mohou být mezi sebou v různých strukturálních vztazích. V této práci se používají dva vztahy:

- **AD** – vztah předek – potomek (zkratka pochází z anglických výrazů ancestor – descendant)
- **PC** – vztah rodič – přímý potomek (zkratka pochází z anglických výrazů parent – child)

Tyto vztahy odpovídají XPath osám, které byly popsány v kapitole 2.2, tabulce 2.1. Vzhledem k tomu, že vztah AD nebo PC značí obousměrný vztah mezi uzly, v případě XPath os se vždy jedná o dvojici os. Vztah AD tedy zastupuje XPath osy `ancestor` a `descendant` a vztah PC zastupuje XPath osy `parent` a `child`. Vztahů mezi uzly existuje více, ale v této práci se bude pracovat pouze se vztahy AD a PC. Vynechání ostatních vztahů je v literatuře běžné, protože tyto osy se v XPath a XQuery používají nejčastěji.

Příkladem dvou uzlů, které jsou ve vztahu PC, mohou být v případě XML dokumentu na obrázku 2.2 uzly $\mathtt{a}_1$ a $\mathtt{b}_1$. Uzel $\mathtt{b}_1$ je v tomto případě přímým potomkem uzlu $\mathtt{a}_1$. Uzel $\mathtt{c}_1$ je také potomkem uzlu $\mathtt{a}_1$, ale není jeho přímým potomkem, tyto uzly jsou tedy ve vztahu AD. Platí, že každé dva uzly, které jsou ve vztahu PC, jsou zároveň ve vztahu AD. Uzly $\mathtt{a}_1$ a $\mathtt{b}_1$ jsou tedy zároveň ve vztahu AD.

2.6 Značka uzlu

Každému uzlu v XML stromu je určena unikátní značka. Pomocí této značky lze vyhodnocovat vztahy mezi uzly. Existuje několik druhů značkovacích schémat. V této práci se používá systém značkování *Containment* [4]. U tohoto systému se značka uzlu skládá ze čtyř částí: $(doc\_id, left, right, level)$. *doc_id* specifikuje identifikátor XML dokumentu. Jelikož se v této práci pracuje vždy pouze s jedním dokumentem, bude tato hodnota ignorována. Po *doc_id* následují hodnoty $left$, $right$ a $level$. Pro určení těchto hodnot se používá postup znázorněný v Algoritmu 1.

Porovnáním hodnot $left$ a $right$ lze určit, zda jsou dva uzly ve vztahu AD. Pokud interval $(n_2.left, n_2.right)$ uzlu n_2 leží uvnitř intervalu $(n_1.left, n_1.right)$ uzlu n_1, uzel n_2 je potomkem uzlu n_1. Pokud se navíc porovnají i hodnoty $level$, lze určit, zda jsou tyto uzly ve vztahu PC. Uzel n_1 je tedy přímým rodičem uzlu n_2, pokud interval $(n_1.left, n_1.right)$ uzlu n_1 obsahuje interval $(n_2.left, n_2.right)$ uzlu n_2 a $n_1.level + 1 = n_2.level$.

Algoritmus 1: Přiřazení značek jednotlivým uzlům

Data: labelCounter – statická proměnná inicializována na hodnotu 0
Vstup: node – kořenový uzel
level – proměnná určující úroveň, na které se uzel node nachází (na začátku nastavena na hodnotu 1)

```
1 Function AssignLabels(node, level):
2 |   node.left = ++labelCounter;
3 |   node.level = level;
4 |   foreach childNode in node.getChildren() do
5 |   |   AssignLabels(childNode, level + 1);
6 |   end
7 |   node.right = ++labelCounter;
8 end
```

Kapitola 3

Algoritmy binárních spojení

Existuje více přístupů ke zpracování TPQ [5]. Jedním z nich je přístup založený na binárních spojeních. Tento přístup rozkládá TPQ do několika algoritmů – binárních spojení. Pořadí těchto binárních spojení tvoří plán vykonání dotazu. Tímto přístupem tedy vznikne několik operátorů, kde výstup jednoho operátoru je jedním ze dvou vstupů následujícího operátoru. Dalším přístupem ke zpracování TPQ je holistický přístup. U tohoto přístupu se na rozdíl od binárního přístupu používá pouze jeden operátor holistického spojení. Dále existují také přístupy, u kterých se výše zmíněné přístupy kombinují.

Výše zmíněné přístupy byly rozděleny dle použitých operátorů. Přístup ke zpracování TPQ se dá rozdělit i podle přístupu k jednotlivým uzlům. Opět existuje více přístupů. Jedním z nich je přístup založený na slévání. V takovém případě existuje několik seznamů, kde každý seznam obsahuje XML uzly se stejným názvem. Tyto seznamy se poté slévají do dvojic tak, aby mezi uzly v těchto dvojicích platil určený vztah. Dalším přístupem je přístup založený na navigaci. V takovém případě se vyhledávání jednotlivých XML uzlů provádí prohledáváním podstromů jiných uzlů. Stejně jako u rozdělení v předchozím odstavci existují také přístupy, u kterých se výše zmíněné přístupy kombinují.

Tato práce se z hlediska přístupu k jednotlivým uzlům zaměřuje na přístup založený na slévání. Z hlediska použitých operátorů se práce zaměřuje na přístup založený na binárních spojeních. Základní rysy algoritmů binárních spojení jsou postupně popsány v této kapitole.

3.1 Tree-Merge Join

Algoritmus *Tree-Merge Join* vychází z tradičního algoritmu spojení z relačních databází – *Merge Join*. Rozšíření spočívá v tom, že se místo porovnávání rovnosti dvou prvků, jako je tomu u relační verze tohoto algoritmu, porovnává nerovnost na základě značky uzlu, která je popsána v kapitole 2.6.

Algoritmus *TreeMergeDesc* pro vztah *AD* je znázorněn v Algoritmu 2. Vstupem algoritmu jsou dva seznamy prvků `listA` a `listD`, kde seznam `listA` obsahuje potenciální předky a seznam `listD`

potenciální potomky. Tyto vstupy jsou u všech algoritmů binárních spojení popisovaných v této práci stejné a při popisu použití těchto algoritmů bude nadále pro přehlednost seznam `listA` adresován jako *levý* vstup a seznam `listD` jako *pravý* vstup. Na rozdíl od relačního *Merge-Join* algoritmu, u kterého se prvky levého i pravého operátoru procházejí pouze jednou, se může stát, že se prvky pravého operátoru budou procházet několikrát. Výstup je seřazen podle operátoru, který byl procházen vnějším cyklem (řádky 3 – 15). Algoritmus *TreeMergeDesc* pro vztah *PC* je stejný jako algoritmus pro vztah *AD*, akorát se musí rozšířit podmínka na řádku 9 o kontrolu, zda je aktuálně procházený prvek ze seznamu `listD` přímým předkem aktuálně procházeného prvku ze seznamu `a`. Tato podmínka je znázorněna v Algoritmu 3.

Algoritmus 2: TreeMergeDesc algoritmus pro vztah *AD*

Vstup: seznamy prvků listA a listD, kde listA obsahuje potenciální předky a seznam listD potenciální potomky a prvky v těchto seznamech jsou seřazené podle atributu left
Výstup: Seznam dvojic $[a_i, d_j]$, kde $a_i \in listA$ a $d_j \in listD$ a prvky a_i a d_j jsou ve vztahu *AD*

```
1  output = prázdný seznam dvojic;
2  beginListA = listA.GetCurrentIterator();
3  while listD.current ≠ NULL do
4  |  while beginListA.current ≠ NULL && beginListA.current.right < listD.current.left do
5  |  |  beginListA.moveNext();
6  |  end
7  |  a = beginListA.GetCurrentIterator();
8  |  while a.current ≠ NULL && a.current.left < listD.current.left do
9  |  |  if a.current.left < listD.current.left && listD.current.right < a.current.right then
10 |  |  |  output.push(a.current, listD.current);
11 |  |  end
12 |  |  a.moveNext();
13 |  end
14 |  listD.moveNext();
15 end
```

Algoritmus 3: Upravující podmínka algoritmu TreeMergeDesc pro vztah *PC*

```
1 if a.current.left < listD.current.left && listD.current.right < a.current.right &&
    listD.current.level = a.current.level + 1 then
2 |  output.push(a.current, listD.current);
3 end
```

3.2 Stack-Tree Join

Stack-Tree algoritmus spojení je oproti *Tree-Merge* algoritmu výhodnější, protože vstupní prvky se nemusí procházet víckrát, stačí pouze jeden průchod. Této vlastnosti je docíleno použitím zásobníku, který je v průběhu algoritmu plněn XML uzly, které leží na cestě mezi kořenovým uzlem a uzlem na vrcholu zásobníku. Je zajištěno, že potomek se v zásobníku vždy nachází výše než jeho předek. Na základě experimentů lze říci, že *Stack-Tree* je obecně efektivnější než *Tree-Merge* [6].

3.2.1 StackTreeDesc

Vstupem *StackTreeDesc* algoritmu jsou dva seznamy prvků `listA` a `listD`, kde seznam `listA` obsahuje potenciální předky a seznam `listD` potenciální potomky. Výstupem jsou všechny dvojice $[a_i, d_j]$, kde $a_i \in listA$ a $d_j \in listD$ a prvky a_i a d_j jsou ve vztahu AD nebo PC. Výstup je seřazen lexikograficky podle (d_j, a_i).

StackTreeDesc pro vztah AD Algoritmus *StackTreeDesc* pro vztah AD je znázorněn v Algoritmu 4. Na začátku vykonávání algoritmu se na řádcích 1 a 2 inicializuje prázdný zásobník `stack` a prázdný seznam `output`, do kterého se budou postupně vkládat výsledné dvojice. Poté se na řádcích 3 a 4 iterátory vstupních seznamů `listA` a `listD` posunou na první prvek. Následuje cyklus, který se provádí tak dlouho, dokud vstupní seznamy nebo zásobník nejsou prázdné (řádky 5 – 17). Pokud je aktuálně procházený prvek ze seznamu `listA` předkem aktuálně procházeného prvku ze seznamu `listD`, vloží se na vrchol zásobníku a `listA` se posune na další prvek (řádky 8 – 10). Poté, co se iterátor `listA` posune na prvek, který není předkem aktuálně procházeného prvku ze seznamu `listD`, projdou se jednotlivé prvky v zásobníku od posledního prvku po vrchol a do seznamu `output` se vloží dvojice, která se skládá z aktuálně procházeného prvku zásobníku a aktuálně procházeného prvku ze seznamu `listD` (řádky 11 – 14). Poté se na řádku 15 posune iterátor seznamu `listD` na další prvek. Pokud v některém z dalších průchodů cyklu je prvek na vrcholu zásobníku předchůdcem obou aktuálně procházených prvků vstupních seznamů, prvek z vrcholu zásobníku se odstraní (řádky 6 – 7). Jakmile iterátory vstupních seznamů projdou všechny prvky a zásobník je prázdný, cyklus se ukončí a algoritmus vrátí na výstup seznam všech dvojic $[a_i, d_j]$, kde a_i je ze seznamu `listA`, d_j ze seznamu `listD` a prvky a_i a d_j jsou ve vztahu AD.

StackTreeDesc pro vztah PC Algoritmus *StackTreeDesc* pro vztah PC je podobný algoritmu pro vztah AD. Jediný rozdíl je v kroku, ve kterém se přidávají dvojice prvků do výstupního seznamu. Tento rozdíl je znázorněn v Algoritmu 5. Touto částí kódu je potřeba nahradit řádky 12 – 14 v Algoritmu 4.

V předchozím algoritmu se prošel celý zásobník, protože obsahoval všechny předky aktuálně procházeného prvku seznamu `listD`. V tomto algoritmu je ale požadován pouze prvek, který je přímým rodičem a proto se místo cyklu, ve kterém se procházel celý zásobník, pouze zkontroluje

Algoritmus 4: StackTreeDesc algoritmus pro vztah AD

Vstup: seznamy prvků listA a listD, kde listA obsahuje potenciální předky a seznam listD potenciální potomky a prvky v těchto seznamech jsou seřazené podle atributu left

Výstup: Seznam dvojic $[a_i, d_j]$, kde $a_i \in listA$ a $d_j \in listD$ a prvky a_i a d_j jsou ve vztahu AD

```
1  stack = prázdný zásobník;
2  output = prázdný seznam dvojic;
3  listA.begin();
4  listD.begin();
5  while vstupní seznamy nejsou prázdné nebo zásobník není prázdný do
6      if listA.current.left > stack.top.right && listD.current.left > stack.top.right then
7          stack.pop();
8      else if listA.current.left < listD.current.left then
9          stack.push(listA.current);
10         listA.moveNext();
11     else
12         foreach element a from stack do
13             output.push(a, listD.current);
14         end
15         listD.moveNext();
16     end
17 end
```

prvek na vrcholu zásobníku a pokud platí, že tento prvek je o jednu úroveň výš než aktuálně procházený prvek seznamu `listD`, znamená to, že je jeho přímým rodičem a tato dvojice se zapíše do seznamu `output`.

Algoritmus 5: Upravující podmínka algoritmu StackTreeDesc pro vztah *PC*

```
1 if stack.top.level + 1 == listD.current.level then
2 |   output.push(stack.top, listD.current);
3 end
```

3.2.2 StackTreeAnc

Algoritmus *StackTreeAnc* produkuje stejný výstup jako algoritmus *StackTreeDesc*, ale na rozdíl od *StackTreeDesc*, u kterého je výstup seřazen lexikograficky podle (d_j, a_i), je výstup seřazen podle (a_i, d_j). K výsledku *StackTreeAnc* algoritmu se lze dopracovat i pomocí algoritmu *StackTreeDesc* a následným seřazením výstupu, tento postup je ale výpočetně i prostorově náročnější.

Princip algoritmu *StackTreeAnc* je podobný algoritmu *StackTreeDesc*. Na začátku se inicializuje zásobník `stack`, výstupní seznam `output` a v cyklu se prochází jednotlivé prvky. Rozdíl oproti *StackTreeDesc* je v místě, ve kterém se vkládají dvojice do výstupního seznamu. Když je nalezena nějaká dvojice (a_i, d_j), nemůže být ihned vložena do výstupního seznamu, protože může existovat jiná dvojice (a'_i, d'_j), která ještě nebyla nalezena a která v lexikografickém uspořádání předchází dvojici (a_i, d_j). Z tohoto důvodu jsou v tomto algoritmu ke každému prvku v zásobníku přiřazeny dva seznamy – `self` a `inherit`. Tyto seznamy umožní odklad uložení dvojice na výstup do té doby, než jsou nalezeny všechny další dvojice, které tuto dvojici předchází.

StackTreeAnc pro vztah AD Algoritmus *StackTreeAnc* pro vztah *AD* je znázorněn v Algoritmu 6. Větev, ve které se aktuálně procházený prvek ze seznamu `listA` vkládá do zásobníku (řádky 14 – 16), je stejná jako u *StackTreeDesc* (řádky 8 – 10 v Algoritmu 4). Rozdíl je ve zbývajících dvou větvích. V části kódu, ve které se prochází zásobník a výsledné dvojice se vkládají do výstupního seznamu (řádky 17 – 26), byla přidána podmínka. Pokud se aktuálně procházený prvek zásobníku `stack` nachází na spodku zásobníku, dvojice tvořená tímto prvkem a aktuálně procházeným prvkem seznamu `listD` je ihned vložena na výstup (řádky 19 – 20). V opačném případě je tato dvojice vložena do seznamu `listSelf` patřící aktuálně procházenému prvku ze zásobníku (řádek 22). Tyto dvojice budou na výstup vloženy poté, co všechny dvojice, které tuto dvojici lexikograficky předchází, byly vloženy do výstupního seznamu. V části kódu, ve které se původně pouze odebíral prvek z vrcholu zásobníku, se navíc kontroluje, zda je zásobník prázdný (řádek 6). Pokud ano, všechny dvojice ze seznamu `listInherit` tohoto prvku se vloží na výstup (řádky 7 – 9). V opačném případě se dvojice ze seznamu `listInherit` tohoto prvku vloží do seznamu `listSelf` téhož prvku a tento seznam se poté vloží do seznamu `listInherit` patřící prvku na vrcholu zásobníku (řádky 10 – 13).

Algoritmus 6: StackTreeAnc algoritmus pro vztah AD

Vstup: seznamy prvků listA a listD, kde listA obsahuje potenciální rodiče a seznam listD potenciální potomky a prvky v těchto seznamech jsou seřazené podle atributu left

Výstup: Seznam dvojic $[a_i, d_j]$, kde $a_i \in listA$ a $d_j \in listD$ a prvky a_i a d_j jsou ve vztahu PC

```
1  output = prázdný seznam dvojic;
2  stack = prázdný zásobník;
3  while vstupní seznamy nejsou prázdné nebo zásobník není prázdný do
4      if listA.current.left > stack.top.right && listD.current.left > stack.top.right then
5          item = stack.pop();
6          if stack.isEmpty() then
7              foreach pair pair from item.listInherit do
8                  output.push(pair);
9              end
10         else
11             item.listSelf.push(item.listInherit);
12             stack.top.listInherit.push(item.listSelf);
13         end
14     else if listA.current.left < listD.current.left then
15         stack.push(listA.current);
16         listA.moveNext();
17     else
18         foreach element a from stack do
19             if a == stack.bottom then
20                 output.push(a, listD.current);
21             else
22                 a.listSelf.push(a, listD.current)
23             end
24         end
25         listD.moveNext();
26     end
27 end
```

Vztah	Třídění a filtrování dle	
	předka	**potomka**
AD	*SemiJoinAncAD*	*SemiJoinDescAD*
PC	*SemiJoinAncPC*	*SemiJoinDescPC*

Tabulka 3.1: Srovnání *SemiJoin* algoritmů

StackTreeAnc pro vztah PC Algoritmus *StackTreeAnc* pro vztah *PC* je opět skoro stejný jako pro vztah *AD*. Rozdíl je v řádcích 18 – 24, kde se cyklus, který prochází celý zásobník, nahradí stejnou upravující podmínkou, která byla použita pro úpravu algoritmu *StackTreeDescPC* a je znázorněna v Algoritmu 5.

3.3 SemiJoin

SemiJoin algoritmy jsou speciální kategorií *StackTree* algoritmů a umožňují efektivnější zpracování dotazu. Na rozdíl od *StackTree* algoritmů, u kterých jsou výstupem všechny dvojice $[a_i, d_j]$, kde $a_i \in listA$ a $d_j \in listD$ a prvky a_i a d_j jsou ve vztahu *AD* nebo *PC*, u *SemiJoin* algoritmů platí, že výstupem je pouze jeden prvek z této dvojice. Zároveň platí, že výsledek je seřazen podle těch prvků, které jsou na výstupu. Podle toho, který prvek bude na výstupu, se rozlišuje mezi dvěma algoritmy – *SemiJoinAnc* a *SemiJoinDesc*. Oproti *StackTree* algoritmům, u kterých je výběr vztahu mezi uzly (*AD* nebo *PC*) stanoven jako parametr algoritmu, tzn. pro oba vztahy se používá stejný algoritmus, u *SemiJoin* algoritmů jsou tyto vztahy řešeny samostatným algoritmem. To je dáno tím, že zpracování dotazu pro vztah *AD* je efektivnější než pro vztah *PC*, protože pro vztah *AD* není třeba použít žádný zásobník. Jedná se tedy o 4 algoritmy – *SemiJoinAncAD*, *SemiJoinAncPC*, *SemiJoinDescAD*, *SemiJoinDescPC*. Použití těchto algoritmů je shrnuto v Tabulce 3.1.

Algoritmus *SemiJoinAncAD* je znázorněn v Algoritmu 7. Vstupem algoritmu jsou dva seznamy `listA` a `listD`. Tento algoritmus provádí spojení uzlů na základě vztahu *AD* a na výstup vkládá uzly z levého vstupního seznamu (zkratka *Anc* v názvu algoritmu). V cyklu, který trvá tak dlouho, dokud nejsou vstupní seznamy prázdné, se prochází jednotlivé prvky vstupních seznamů. Pokud je hodnota atributu `left` aktuálně procházeného prvku ze seznamu `listA` větší nebo rovna hodnotě atributu `left` aktuálně procházeného prvku ze seznamu `listD`, znamená to, že prvek ze seznamu `listA` je následovníkem prvku ze seznamu `listD`, tudíž mezi nimi vztah AD není a seznam `listD` se posune na další prvek (řádky 2 a 3). Pokud je hodnota atributu `left` aktuálně procházeného prvku ze seznamu `listA` menší než u aktuálně procházeného prvku ze seznamu `listD` a zároveň je hodnota atributu `right` aktuálně procházeného prvku ze seznamu `listA` větší než hodnota stejnojmenného atributu aktuálně procházeného prvku ze seznamu `listD`, tyto prvky jsou ve vztahu AD a prvek ze seznamu `listA` se vloží na výstup a seznam `listA` se posune na další prvek. Pokud ani jedna

z předchozích dvou podmínek neplatí, prvek ze seznamu `listA` je předchůdcem prvku ze seznamu `listD` a seznam `listA` se posune na další prvek.

Algoritmy *SemiJoinAncPC* a *SemiJoinDescPC* vychází z algoritmů *StackTreeAnc* a *StackTreeDesc*. Algoritmus *SemiJoinDescAD* je obdobou algoritmu *SemiJoinAncAD*. Vzhledem k tomu, že úpravy jsou přímočaré, nebudou tyto algoritmy popsány.

Algoritmus 7: SemiJoinAncAD algoritmus pro vztah AD

Vstup:
Výstup:

```
1  while seznamy listA a listD nejsou prázdné do
2      if listA.current.left ≥ listD.current.left then
3          listD.moveNext();
4      else if listA.current.right > listD.current.right then
5          output.push(listA.current);
6          listA.moveNext();
7      else
8          listA.moveNext();
9      end
10 end
```

Kapitola 4

Algoritmy pro sestavení plánů

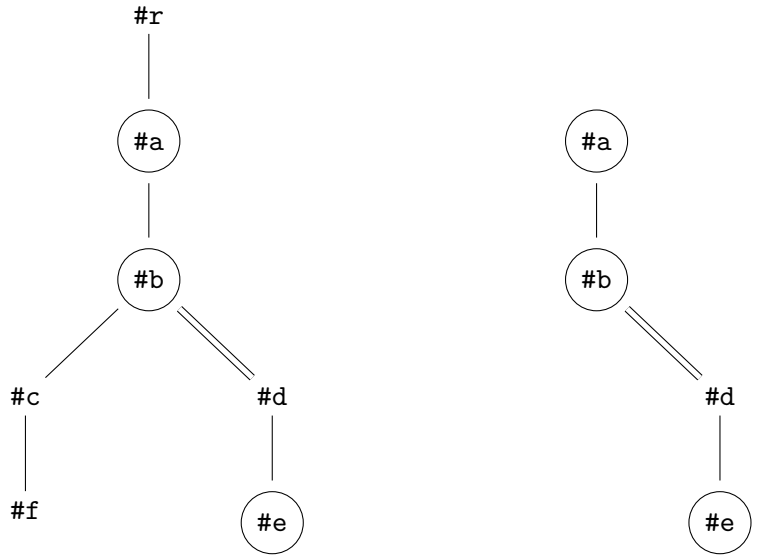
V kapitole 3 byly popsány algoritmy binárních spojení, které ale samy o sobě neumí zpracovat TPQ dotaz. K vyhodnocení TPQ dotazu slouží plán vykonání dotazu, který se skládá z vhodně uspořádaných algoritmů binárních spojení. V této kapitole je představeno několik algoritmů, které slouží k sestavení plánu vykonání dotazu. Tyto algoritmy jsou rozděleny do následujících tří kategorií:

- **Algoritmy bez cenové optimalizace** – tyto algoritmy jsou popsány v kapitole 4.2.
- **Algoritmy s cenovou optimalizací** – algoritmy v této kategorii rozšiřují algoritmy bez cenové optimalizace o určení ceny plánu a výběr toho nejlepšího možného plánu. Tyto algoritmy jsou popsány v kapitole 4.3.
- **Algoritmy s heuristickou optimalizací** – algoritmy v této kategorii opět rozšiřují algoritmy bez cenové optimalizace, ale pro výběr plánu se používá heuristika místo ceny. Tyto algoritmy jsou popsány v kapitole 4.4.

Proces sestavení plánu je z velké části ovlivněn konfigurací výstupních a nevýstupních uzlů v dotazu. Z tohoto důvodu se v následujícím textu rozlišují následující tři typy dotazů:

- dotazy s jedním výstupním uzlem,
- dotazy s více výstupními uzly,
- dotazy, ve kterých jsou výstupní uzly všechny.

Tato práce se zaměřuje pouze na plně zřetězené plány, zkráceně FP plány (z angl. *fully-pipelined*). U těchto plánů jednotlivé operátory spojení poskytují výsledek seřazený tak, jak ho potřebují následující operátory. Díky tomu není potřeba nikde v průběhu zpracovávání dotazu třídit mezivýsledky a také není potřeba uchovávat velké množství mezivýsledků. Tyto plány jsou tedy prostorově i časově méně náročné než nezřetězené plány [7].



(a) TPQ (b) Jádro dotazu

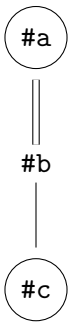
Obrázek 4.1: TPQ a jádro dotazu

Před popisem jednotlivých algoritmů je potřeba zavést dva nové pojmy, a to *jádro dotazu* a *uzel jádra dotazu*. *Uzel jádra dotazu* je takový uzel, který je výstupní nebo se v TPQ nachází mezi dvěma výstupními uzly. *Jádro dotazu* je minimální podstrom TPQ, který obsahuje všechny *uzly jádra dotazu*. Na obrázku 4.1a se nachází příklad TPQ a obrázek 4.1b ukazuje, jak vypadá jádro dotazu pro tento TPQ. Podrobnější popis jádra dotazu můžeme nalézt v literatuře [8].

4.1 Problém uspořádání mezivýsledku

Při zpracování dotazů, které obsahují v jádře dotazu nevýstupní uzly, může dojít k problému, kdy mezivýsledek nebude správně seřazen. Příklad dotazu, ve kterém je uvnitř jádra dotazu nevýstupní uzel a na kterém bude prezentován problém uspořádání mezivýsledku, je znázorněn na obrázku 4.2. Dotaz se vztahuje k dokumentu na obrázku 2.2.

Plán vykonání dotazu, u kterého dochází k problému uspořádání mezivýsledku, je znázorněn na obrázku 4.3a. Aby byl konečný výsledek správně seřazen, je potřeba TPQ zpracovávat odspodu. Tímto způsobem TPQ uzly poskytují mezivýsledek seřazený tak, jak ho potřebují TPQ uzly nad nimi Je tedy potřeba provést spojení uzlů `#b` a `#c`. Použije se algoritmus *StackTreeAnc*, kde levý vstup je seznam uzlů `#b` a pravý vstup je seznam uzlů `#c`. Vztah mezi těmito uzly musí být PC. Výsledek tohoto spojení je zobrazen na obrázku 4.3b, ve kterém je vidět, že je výsledek seřazen podle uzlu `#b`. Tento výsledek je dále potřeba spojit s uzly `#a`. Pokud se pro spojení použije opět algoritmus *StackTreeAnc*, kde levým vstupem je seznam uzlů `#a` a pravým vstupem je výsledek předchozího spojení, výsledkem bude tabulka na obrázku 4.3c. Vzhledem k tomu, že v TPQ je dotazovací uzel



Obrázek 4.2: TPQ s nevýstupním uzlem

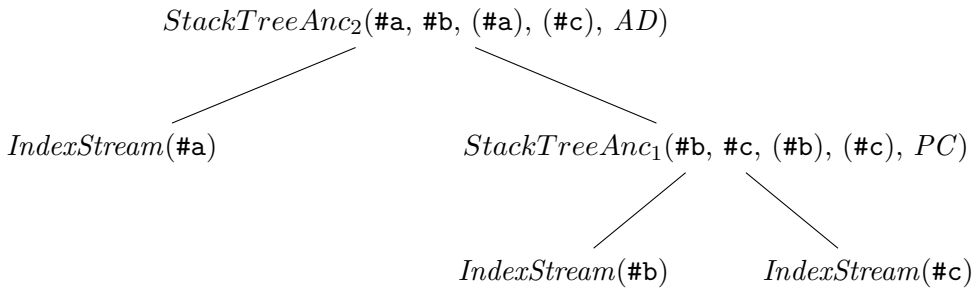
`#b` nevýstupní, sloupec s těmito uzly se z výsledku odstraní. Finální výsledek spojení je zobrazen na obrázku 4.3d, ve kterém je vidět, že uzly `#c` nejsou správně seřazeny.

Tento problém řeší speciální varianta algoritmu *StackTreeAnc* nazvaná *StackTreeAncSrt*, která přijímá jeden vstupní parametr navíc. Tento parametr specifikuje, který uzel se má použít pro tzv. sekundární test vztahu mezi uzly (zkratka *Srt* v názvu algoritmu vychází z anglického výrazu *Secondary relationship test*). Plán vykonání dotazu, ve kterém je použit tento algoritmus a ve kterém je tudíž vyřešen problém uspořádání mezivýsledku, je znázorněn na obrázku 4.4a. Pro spojení uzlů `#b` a `#c` se použije algoritmus *StackTreeDesc*, aby byl výsledek správně seřazen podle uzlu `#c`. Výsledek spojení je zobrazen na obrázku 4.4b. Pro spojení uzlů `#a` s předchozím výsledkem se použije algoritmus *StackTreeAncSrt*, kde levým vstupem je seznam uzlů `#a` a pravým vstupem je výsledek předchozího spojení. Uzel z levého vstupu, který se má použít pro spojení, je opět nastaven na `#a`. Oproti předchozímu příkladu je uzel z pravého vstupu, který se má použít pro spojení, nastaven na `#c`. Parametr pro specifikaci uzlu, který se má použít pro sekundární test vztahu a je uveden jako poslední parametr, je nastaven na `#b`. Algoritmus v tomto příkladu funguje tak, že spojí dva vstupy na základě vztahu AD mezi uzly `#a` a `#c`. Pro tento test vztahu se vždy použije vztah AD, nehledě na to, jaký vztah je u tohoto algoritmu nastaven. Poté se provede sekundární test, ve kterém se zkontroluje vztah mezi uzly `#a` a `#b`. Pro tuto kontrolu se již použije vztah specifikovaný parametrem algoritmu (v tomto případě AD). Tímto postupem se zajistí, že výsledek spojení bude seřazen správně. Mezivýsledek po provedení spojení je zobrazen na obrázku 4.4c. Vzhledem k tomu, že uzel `#b` je nevýstupní, je sloupec s těmito uzly z výsledku odstraněn. Finální výsledek spojení je zobrazen na obrázku 4.4d. Tento výsledek je již uspořádán správně.

4.2 Sestavení plánu bez cenové optimalizace

4.2.1 Dotazy s jedním výstupním uzlem

Algoritmus pro zpracování dotazů s jedním výstupním uzlem je znázorněn v Algoritmu 8. Tento algoritmus byl publikován v článku [8]. Algoritmus bude v této práci označován jako *BinOne*.



(a) Plán vykonání dotazu obsahující problém uspořádání mezivýsledku

#b	#c
b1	c1
b2	c3
b3	c2
b5	c4

(b) Výsledek spojení $StackTreeAnc_1$

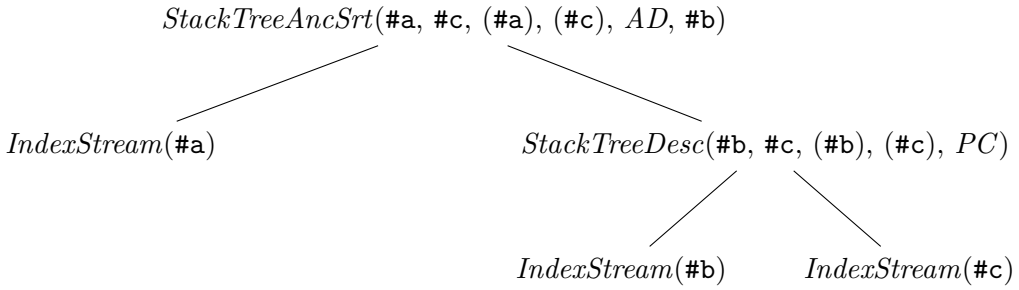
#a	#b	#c
a1	b1	c1
a1	b2	c3
a1	b3	c2
a3	b5	c4

(c) Výsledek spojení $StackTreeAnc_2$ před odstraněním sloupce `#b`

#a	#c
a1	c1
a1	c3
a1	c2
a3	c4

(d) Konečný výsledek spojení $StackTreeAnc_2$

Obrázek 4.3: Příklad zpracování dotazu obsahující problém uspořádání mezivýsledku



(a) Plán vykonání dotazu s vyřešeným problémem uspořádání mezivýsledku

#b	#c
b1	c1
b3	c2
b2	c3
b5	c4

(b) Výsledek spojení *StackTreeDesc*

#a	#b	#c
a1	b1	c1
a1	b3	c2
a1	b2	c3
a3	b5	c4

(c) Výsledek spojení *StackTreeAncSrt* před odstraněním sloupce `#b`

#a	#c
a1	c1
a1	c2
a1	c3
a3	c4

(d) Konečný výsledek spojení *StackTreeAncSrt*

Obrázek 4.4: Příklad zpracování dotazu s vyřešeným problémem uspořádání mezivýsledku

Nejprve je potřeba nalézt výstupní uzel. Tento uzel je poté vstupním parametrem `node` funkce `BuildPlanBinOne`.

Na začátku algoritmu se na řádku 2 vytvoří operátor `IndexStream`, který při spuštění plánu vrací seznam všech XML uzlů s určitým názvem. Tento název je určen parametrem, který je v tomto případě nastaven na vstupní parametr `node`. Vytvořený operátor se vloží do proměnné `plan`. Poté se na řádcích 3 – 18 v cyklu prochází sousední uzly parametru `node`. Pojem sousední uzel je definován v kapitole 2.1.1. Pro každý sousední uzel se na řádku 4 rekurzivně zavolá metoda `BuildPlanBinOne`, která vytvoří plán pro podstrom, kde je kořenovým uzlem daný sousední uzel. Výsledný operátor se vloží do proměnné `neighborPlan`. Poté se tyto dva plány musí spojit pomocí algoritmu binárního spojení. Vzhledem k tomu, že v dotazu je pouze jeden výstupní uzel, je možno pro spojení využít efektivnějších *SemiJoin* algoritmů. Jak již bylo zmíněno v kapitole 3.3, u tohoto typu algoritmů existují samostatné algoritmy jak pro typ filtrování (předek/potomek), tak pro vztahy mezi uzly (AD/PC). Z tohoto důvodu je třeba rozlišit, který algoritmus se musí použít. Toto zjištění se provádí na řádcích 5 – 17. Pokud je sousední uzel rodičem vstupního uzlu, použije se jeden z algoritmů *SemiJoinDescAD* a *SemiJoinDescPC* (řádky 6 – 10). Do algoritmu se jako levý vstup vloží operátor plánu sousedního uzlu `neighborPlan` a jako pravý vstup operátor plánu vstupního uzlu `plan`. Pokud je vztah mezi uzly opačný, tedy sousední uzel je potomkem vstupního uzlu, použije se jeden z algoritmů `SemiJoinAncAD` a `SemiJoinAncPC` (řádky 12 – 16). Zde je pořadí vstupních plánů do algoritmu opačné, tedy jako levý vstup algoritmu se použije operátor `plan` a jako pravý `neighborPlan`. V obou případech platí, že vztah mezi vstupním a sousedním uzlem určí, zda se použije algoritmus pro vztah AD nebo PC (řádky 7, 9, 13, 15). Po zvolení správného algoritmu se operátor výsledného algoritmu vloží do proměnné `plan`. Po projití všech sousedních uzlů vstupního uzlu se cyklus ukončí a v proměnné `plan` je konečný operátor plánu vykonání dotazu.

Na řádku 3 se nachází prostor pro cenovou optimalizaci. V tomto algoritmu se spoléhá na nějaké výchozí pořadí sousedních uzlů, ale volbou pořadí těchto uzlů je možné dosáhnout efektivnějšího zpracování. Přestože se touto volbou pořadí může zvýšit efektivita vykonávání, asymptotická složitost se tím nemění [8].

4.2.2 Dotazy s více výstupními uzly

Algoritmus pro zpracování dotazů s více výstupními uzly je znázorněn v Algoritmu 9. Tento algoritmus bude nadále označován jako *BinMultiple*. Vstupem tohoto algoritmu je uzel jádra dotazu. Pojem *uzel jádra dotazu* je definován na začátku kapitoly 4. Před voláním funkce `BuildPlanBinMultiple` je tedy nejprve nutné nalézt kořenový uzel jádra dotazu a použít ho jako vstup do této funkce.

Na začátku algoritmu se na řádku 2 zavolá funkce `BuildPlanBinOne`. Tato funkce byla popsána v předchozí kapitole a funguje tak, že vytvoří plán vykonání dotazu pro vstupní uzel `node` a všechny jeho okolní uzly, které nejsou uzly jádra dotazu. Výsledný plán se vloží do proměnné `plan`. Poté se zjistí, kteří potomci uzlu `node` jsou uzly jádra dotazu a tyto uzly se v cyklu projdou (řádky 3 – 31).

Algoritmus 8: `BuildPlanBinOne`

Vstup: výstupní dotazovací uzel node
Výstup: operátor plánu vykonání dotazu

```
1  Function BuildPlanBinOne(node):
2      plan = IndexStream(node);
3      foreach neighbor in node.getNeighbors() do
4          neighborPlan = BuildPlanBinOne(neighbor);
5          if neighbor is parent of node then
6              if relationship between neighbor and node is AD then
7                  plan = SemiJoinDescAD(neighborPlan, plan);
8              else if relationship between neighbor and node is PC then
9                  plan = SemiJoinDescPC(neighborPlan, plan);
10             end
11         else if node is parent of neighbor then
12             if relationship between node and neighbor is AD then
13                 plan = SemiJoinAncAD(plan, neighborPlan);
14             else if relationship between node and neighbor is PC then
15                 plan = SemiJoinAncPC(plan, neighborPlan);
16             end
17         end
18     end
19     return plan;
20 end
```

U každého potomka, který je uzlem jádra dotazu, se na řádku 4 zjistí, zda je tento uzel výstupní. Pokud ano, vytvoří se pro tento uzel plán vykonání pomocí rekurzivního volání funkce `BuildPlanBinMultiple` (řádek 5). Výsledný plán se uloží do proměnné `subPlan`. Poté se na řádku 9 plány `plan` a `subPlan` spojí pomocí algoritmu *StackTreeAnc*.

Pokud aktuálně procházený uzel není výstupní (řádek 10), je potřeba projít všechny nevýstupní uzly, které se nachází mezi aktuálně procházeným uzlem a výstupním uzlem, který se v TPQ stromu nachází hlouběji. Nejprve se tedy pro aktuálně procházený uzel vytvoří plán pomocí funkce `BuildPlanBinOne` (řádek 11). Poté se na řádcích 13 – 25 provádí postupné zanořování přes potomky, dokud se nenarazí na výstupní uzel. Je zaručeno, že uzel jádra dotazu, který není výstupní, má vždy přesně jednoho potomka, který vede k výstupnímu uzlu [8]. Pro každý z těchto procházených potomků se na řádcích 14 – 24 vytvoří plány vykonání a spojí se s dříve vytvořeným plánem `subPlan`. Takto se postupně vytvoří plán pro celou větev nevýstupních uzlů jádra dotazu. Pro spojování plánů se na řádku 23 použije algoritmus *StackTreeDesc* a výsledný plán se vloží do proměnné `subPlan`. Po projití všech nevýstupních uzlů jádra dotazu v této větvi TPQ stromu a nalezení výstupního uzlu se na řádku 29 spojí dříve vytvořený plán v proměnné `plan` a plán pro větev s nevýstupními uzly jádra dotazu, který je uložen v proměnné `subPlan`. Pro spojení je třeba použít algoritmus *StackTreeAncSrt*. Tato speciální varianta algoritmu *StackTreeAnc* je potřeba z důvodu problému uspořádání mezivýsledku, který je popsán v kapitole 4.1. Výsledný plán se vloží do proměnné `plan`.

Na konci algoritmu je v proměnné `plan` výsledný plán vykonání dotazu pro dotazy s více výstupními uzly.

4.3 Sestavení plánu s cenovou optimalizací

Obecným principem cenové optimalizace je vytvoření množiny kandidátních plánů vykonání, u kterých se následně určí cena. Nakonec se z této množiny vybere plán, který je nejlevnější. Vzhledem k tomu, že všech možných plánů může vzniknout velké množství, je potřeba vhodně vybírat jen ty kandidátní plány, u kterých se předpokládá, že budou nějakým způsobem efektivní. Algoritmy v této kapitole rozšiřují algoritmy uvedené v kapitole 4.2. Rozšíření spočívá ve schopnosti určit cenu jakéhokoliv plánu a následně vytvoření kandidátních plánů, ze kterých se vybere ten nejlevnější. U ceny plánu platí, že plán s nižší cenou je efektivnější než plán s vyšší cenou. Způsob stanovení cenového modelu je podrobněji popsán v kapitole 5.1.

4.3.1 Dotazy se všemi výstupními uzly

Algoritmus pro zpracování dotazů se všemi výstupními uzly s cenovou optimalizací je znázorněn v Algoritmech 10 a 11. V Algoritmu 10 se nachází počáteční funkce, ve které se poté volá funkce znázorněna v Algoritmu 11. Tento algoritmus vychází z článku [7] a zpracovává pouze takové dotazy,

Algoritmus 9: `BuildPlanBinMultiple`

Vstup: node – první uzel jádra dotazu
Výstup: plán vykonání dotazu

```
1  Function BuildPlanBinMultiple(node):
2      plan = BuildPlanBinOne(node);
3      foreach coreChild in node.getCoreChildren() do
4          if coreChild is output then
5              subPlan = BuildPlanBinMultiple(coreChild);
6              relationship = GetRelationship(node, coreChild);
7              ancOutput = all nodes in plan;
8              descOutput = all nodes in subPlan;
9              plan = StackTreeAnc(plan, subPlan, node, coreChild, ancOutput, descOutput,
                relationship);
10         else
11             subPlan = BuildPlanBinOne(coreChild);
12             nextChild = coreChild;
13             while nextChild is not output do
14                 coreChildOfNextChild = nextChild.getCoreChild();
15                 if coreChildOfNextChild is not output then
16                     subSubPlan = BuildPlanBinOne(coreChildOfNextChild);
17                 else
18                     subSubPlan = BuildPlanBinMultiple(coreChildOfNextChild);
19                 end
20                 relationship = GetRelationship(nextChild, coreChildOfNextChild);
21                 ancOutput = coreChild;
22                 descOutput = all nodes in subSubPlan;
23                 subPlan = StackTreeDesc(subPlan, subSubPlan, nextChild,
                    coreChildOfNextChild, ancOutput, descOutput, relationship);
24                 nextChild = coreChildOfNextChild;
25             end
26             relationship = GetRelationship(node, coreChild);
27             ancOutput = all nodes in plan;
28             descOutput = all nodes in subPlan except coreChildOfNextChild;
29             plan = StackTreeAncSrt(plan, subPlan, node, nextChild, ancOutput, descOutput,
                relationship, coreChild);
30         end
31     end
32     return plan;
33 end
```

ve kterých jsou všechny uzly výstupní. V následujícím textu bude tento algoritmus nadále označován jako *BinAllCost*.

Tento algoritmus je podobný algoritmu pro zpracování dotazů s jedním výstupním uzlem s tím rozdílem, že pro spojení jednotlivých plánů se nepoužívá algoritmus typu *SemiJoin*, ale jeden z algoritmů typu *StackTree*. Tyto algoritmy byly popsány v kapitolách 3.3 a 3.2. Dalším rozdílem je zakomponování cenové optimalizace.

V Algoritmu 10 je znázorněna první část tohoto algoritmu. Vzhledem k tomu, že v dotazech, které zpracovává tento algoritmus, jsou všechny uzly výstupní, je možno tvoření plánu začít jakýmkoliv uzlem v TPQ stromu. Zde se nachází první příležitost pro uplatnění cenové optimalizace. Spočívá v tom, že algoritmus projde všechny uzly v TPQ stromu a vytvoří pro ně plán. Tím, že se bude plán vždy rozvíjet od jiného uzlu, vzniknou plány s různou cenou. Na začátku algoritmu se na řádku 2 inicializuje proměnná `minCost` na největší možnou hodnotu. Na řádku 3 se definuje proměnná `bestPlan`, která je bez hodnoty. Poté se v cyklu projdou jednotlivé uzly v TPQ stromu (řádky 4 – 11). Pro každý uzel se pomocí volání funkce `BuildPlanBinAllCost` vytvoří plán vykonání dotazu a výsledek se uloží do proměnné `plan` (řádek 5). Následně se na řádku 6 spočítá cena plánu a tato cena se na řádku 7 porovná s minimální nalezenou cenou. Pokud je cena plánu nižší, nastaví se proměnná `minCost` na tuto hodnotu a do proměnné `bestPlan` se vloží aktuální plán (řádky 8 a 9). Po projití všech uzlů se v proměnné `bestPlan` nachází nejlevnější možný plán, který je výstupem tohoto algoritmu.

V Algoritmu 11 je znázorněna funkce, která se volá z funkce popsané v předchozím odstavci. Vstupním parametrem je uzel TPQ stromu `node`. Stejně jako u algoritmu s jedním výstupním uzlem, který je popsán v kapitole 4.2.1, se na začátku na řádku 2 vloží do proměnné `plan` operátor `IndexStream` pro vstupní parametr `node`. Na tomto místě se v algoritmu s jedním výstupním uzlem postupně prošly sousední uzly vstupního uzlu `node`, pro každý sousední uzel se vytvořil plán a ten se spojil s doposud vytvořeným plánem. Zde se nachází druhá příležitost pro cenovou optimalizaci. Jednotlivé plány sousedních uzlů mají různou cenu a pořadí spojení těchto plánů s dosavadním plánem ovlivňuje cenu výsledného plánu.

Na řádku 3 se inicializuje proměnná `subPlans`. Jedná se o pole dvojic $(uzel, plán)$. Na řádcích 4 – 6 se v cyklu projdou sousední uzly vstupního uzlu `node`. Pojem sousední uzel je popsán v kapitole 2.1.1. Pro každý sousední uzel se na řádku 5 vytvoří plán vykonání pomocí rekurzivního volání funkce `BuildPlanBinAllCost`. Ze sousedního uzlu a nově vytvořeného plánu se vytvoří dvojice a ta se vloží do pole `subPlans`. Po projití sousedních uzlů se v poli `subPlans` nachází plány pro jednotlivé uzly. Nyní je potřeba najít to nejlepší možné pořadí spojení těchto plánů. To se provede tak, že se vytvoří všechny možné permutace těchto plánů, tyto permutace se postupně projdou a jednotlivé plány se spojí v pořadí určené permutací.

Stejně jako v předchozí funkci se na řádcích 7 a 8 inicializuje proměnná `minCost` a `bestPlan`. Na řádku 9 se inicializuje proměnná `permutations`, do které se vloží všechny permutace prvků z pole `subPlans`. Následně se na řádcích 10 – 33 tyto permutace v cyklu projdou. Na začátku tvoření plánu

pro jednotlivé permutace se na řádku 11 inicializuje proměnná `joinPlan`, která se nastaví na proměnnou `plan`. Poté se na řádcích 12 – 28 začnou procházet jednotlivé dvojice (*sousední uzel*, *plán*) v permutaci. Pro každý sousední uzel se provede spojení jeho plánu s plánem v proměnné `joinPlan`. Pokud je sousední uzel rodičem vstupního uzlu `node`, použije se algoritmus *StackTreeDesc* (řádky 14 – 17). Pokud je sousední uzel potomkem uzlu `node`, použije se algoritmus *StackTreeAnc* (řádky 19 – 22). Vzhledem k tomu, že tento algoritmus zpracovává dotazy se všemi výstupními uzly, není třeba brát v úvahu algoritmus spojení `StackTreeAncSrt`, protože k problému uspořádání mezivýsledku, který je popsán v kapitole 4.1, nikdy nedojde. Na řádcích 14 a 19 se volá metoda `GetRelationship`, která vrací vztah mezi uzly `node` a `neighbor`. Po vytvoření spojení se na řádku 24 vypočítá cena plánu a na řádku 25 se tato cena porovná s minimální nalezenou cenou. Pokud se zjistí, že cena plánu již překročila minimální nalezenou cenu, může být tvoření plánu pro tuto permutaci zastaveno, protože cena plánu pro tuto permutaci bude už jenom horší. V takovém případě se může přejít na další permutaci a ušetří se tím čas tvoření plánu, který se stejně určitě nepoužije. Pokud tato situace nenastane, pokračuje se s dalším sousedním uzlem a v proměnné `joinPlan` se postupně tvoří celkový plán. Po projití všech prvků v permutaci se na řádku 29 zkontroluje, zda je cena aktuálního plánu nižší než minimální nalezená cena. Pokud ano, aktualizuje se proměnná `minCost` na hodnotu ceny plánu a do proměnné `bestPlan` se vloží tento plán (řádky 30 a 31). Po projití všech permutací se v proměnné `bestPlan` nachází nejlepší plán, který má ze všech možných plánů nejnižší cenu. Tento plán je ale pouze nejlepší pro vstupní uzel `node`. Konečný nejlepší plán je vybrán ve funkci `BuildBestPlanBinAllCost`.

Algoritmus 10: `BuildBestPlanBinAllCost`

Vstup: tpq – Twig Pattern Query strom
Výstup: nejlepší možný plán vykonání dotazu

```
1  Function BuildBestPlanBinAllCost(tpq):
2      minCost = ∞;
3      bestPlan = NULL;
4      foreach node in tpq do
5          plan = BuildPlanBinAllCost(node);
           /* definice funkce BuildPlanBinAllCost se nachází v Algoritmu 11    */
6          CalculateCost(plan);
7          if plan.cost < minCost then
8              minCost = plan.cost;
9              bestPlan = plan;
10         end
11     end
12     return bestPlan;
13 end
```

Algoritmus 11: `BuildPlanBinAllCost`

Vstup: node – uzel z TPQ stromu
Výstup: plán vykonání dotazu rozvinutý od uzlu node

```
1  Function BuildPlanBinAllCost(node):
2      plan = IndexStream(node);
3      subPlans = empty array of tuples (neighbor, subPlan);
4      foreach neighbor in node.getNeighbors() do
5          subPlans.add(neighbor, BuildPlanBinAllCost(neighbor));
6      end
7      minCost = ∞;
8      bestPlan = NULL;
9      permutations = all permutations of subPlans;
10     foreach permutation in permutations do
11         joinPlan = plan;
12         foreach tuple (neighbor, subPlan) in permutation do
13             if neighbor is parent of node then
14                 relationship = GetRelationship(neighbor, node);
15                 ancOutput = all nodes in subPlan;
16                 descOutput = all nodes in joinPlan;
17                 joinPlan = StackTreeDesc(subPlan, joinPlan, neighbor, node, ancOutput,
                       descOutput, relationship);
18             else if node is parent of neighbor then
19                 relationship = GetRelationship(node, neighbor);
20                 ancOutput = all nodes in joinPlan;
21                 descOutput = all nodes in subPlan;
22                 joinPlan = StackTreeAnc(joinPlan, subPlan, node, neighbor, ancOutput,
                       descOutput, relationship);
23             end
24             CalculateCost(joinPlan);
25             if joinPlan.cost > minCost then
26                 vytváření plánu pro tuto permutaci může být zastaveno, pokračuje se další
                       permutací;
27             end
28         end
29         if joinPlan.cost < minCost then
30             minCost = joinPlan.cost;
31             bestPlan = joinPlan;
32         end
33     end
34     return bestPlan;
35 end
```

4.3.2 Dotazy s jedním výstupním uzlem

Algoritmus pro zpracování dotazů s jedním výstupním uzlem s cenovou optimalizací je znázorněn v Algoritmu 12. Tento algoritmus bude nadále označován jako *BinOneCost*. Vstupem do tohoto algoritmu je výstupní uzel z TPQ stromu. Princip algoritmu je stejný jako předchozí algoritmus popsaný v kapitole 4.3.1.

Nejprve se na řádku 2 vytvoří operátor *IndexStream* pro vstupní uzel `node`. Poté se na řádcích 3 – 6 vytvoří plány vykonání dotazu pro sousední uzly vstupního uzlu. Následně se na řádku 8 vytvoří všechny možné permutace těchto plánů a na řádcích 9 – 34 se tyto permutace postupně projdou. Poté se pro každou permutaci vytvoří plán vykonání pomocí spojování dosavadního plánu a plánu vykonání pro sousední uzel (řádky 10 – 29). Zde se algoritmus odlišuje od předchozího algoritmu. Vzhledem k tomu, že výstupní uzel je pouze jeden, není třeba používat *StackTree* algoritmy, jejichž výstupem mohou být všechny uzly, ale lze použít *SemiJoin* algoritmy, které jsou efektivnější. Popis těchto algoritmů se nachází v kapitole 3.3. Použití těchto algoritmů pro spojení lze vidět na řádcích 14, 16, 20 a 22. Po každém vytvoření spojení se na řádku 25 spočítá cena plánu a také se zkontroluje, zda cena plánu již nepřekročila minimální nalezenou cenu (řádek 26). Pokud ano, lze vytváření plánu pro tuto permutaci přerušit a pokračovat vytvářením plánu pro další permutaci (řádek 27). Po vytvoření kompletního plánu pro aktuálně procházenou permutaci se na řádku 30 zkontroluje, zda je cena plánu nižší než minimální nalezená cena. Pokud ano, nastaví se proměnné `minCost` a `bestPlan` (řádky 31 a 32) a pokračuje se vytvářením plánu pro další permutaci.

Na konci algoritmu se v proměnné `bestPlan` nachází nejlepší možný plán pro zpracování dotazů s jedním výstupním uzlem. Tento plán se vloží na výstup a algoritmus se ukončí.

4.3.3 Dotazy s libovolným počtem výstupních uzlů

Algoritmus pro zpracování dotazů s libovolným počtem výstupních uzlů s cenovou optimalizací je znázorněn v Algoritmu 13. Tento algoritmus bude nadále označován jako *BinMultipleCost*. Vstupem algoritmu je kořenový uzel jádra dotazu z TPQ stromu dotazu. Princip použití cenové optimalizace je podobný jako v předchozích algoritmech. V místech, kde se spojují plány jednotlivých uzlů, se vyzkouší všechny možné permutace pořadí spojení. Z tohoto důvodu se na začátku algoritmu zavolá funkce `CreateSubPlans`, která vytvoří plány vykonání dotazu pro potomky vstupního uzlu `node`. Tito potomci musí být uzly jádra dotazu. Algoritmus této funkce je znázorněn v Algoritmu 14.

Ve funkci 14 se na řádku 2 vytvoří proměnná `subPlans`, což je prázdné pole dvojic, kde první prvek dvojice je potomek, který je uzlem jádra dotazu, a druhý prvek dvojice je jeho plán vykonání. Poté se na řádcích 3 – 24 projdou potomci vstupního uzlu `node`, kteří jsou uzly jádra dotazu. Pokud je aktuálně procházený prvek výstupní, vytvoří se pro něj plán vykonání pomocí rekurzivního volání funkce `BuildPlanBinMultipleCost` (řádky 4 – 6). Vytvoří se dvojice, kde první prvek je uzel `coreChild` a druhý prvek je zhotovený plán. Tato dvojice se vloží do pole `subPlans`. Pokud aktuálně procházený prvek není výstupní (řádky 6 – 23), provedou se následující kroky. Nejprve se na řádku 7

Algoritmus 12: `BuildPlanBinOneCost`

Vstup: node – výstupní uzel z TPQ stromu
Výstup: nejlepší možný plán vykonání dotazu

```
1  Function BuildPlanBinOneCost(node):
2      plan = IndexStream(node);
3      subPlans = array of tuples (neighbor, subPlan);
4      foreach neighbor in node.getNeighbors() do
5          subPlans.add(BuildPlanBinOneCost(neighbor));
6      end
7      minCost = ∞;
8      permutations = all permutations of subPlans;
9      foreach permutation in permutations do
10         joinPlan = plan;
11         foreach tuple (neighbor, subPlan) in permutation do
12             if neighbor is parent of node then
13                 if relationship between neighbor and node is AD then
14                     joinPlan = SemiJoinDescAD(operatorNeighbor, operatorNode);
15                 else if relationship between neighbor and node is PC then
16                     joinPlan = SemiJoinDescPC(operatorNeighbor, operatorNode);
17                 end
18             else if node is parent of neighbor then
19                 if relationship between node and neighbor is AD then
20                     joinPlan = SemiJoinAncAD(operatorNode, operatorNeighbor);
21                 else if relationship between node and neighbor is PC then
22                     joinPlan = SemiJoinAncPC(operatorNode, operatorNeighbor);
23                 end
24             end
25             CalculateCost(joinPlan);
26             if joinPlan.cost > minCost then
27                 vytváření plánu pro tuto permutaci může být zastaveno, pokračuje se další
                    permutací;
28             end
29         end
30         if joinPlan.cost < minCost then
31             minCost = joinPlan.cost;
32             bestPlan = joinPlan;
33         end
34     end
35     return bestPlan;
36 end
```

vytvoří plán vykonání dotazu pomocí volání funkce `BuildPlanBinOneCost`. Algoritmus této funkce je znázorněn v Algoritmu 12 a je popsán v kapitole 4.3.2. V této funkci je ale potřeba udělat jednu změnu. Změna spočívá v nahrazení řádku 5 kódem specifikovaným v Algoritmu 15. Na řádcích 4 – 6 se procházejí sousední uzly vstupního uzlu a pro každý uzel se vytvoří plán vykonání. Protože je potřeba zpracovávat dotazy, ve kterých může být libovolný počet výstupních uzlů, je nutné na řádek 5 přidat podmínku, která zkontroluje, zda sousední uzel patří do jádra dotazu. Plán získaný z funkce `BuildPlanBinOneCost` se na řádku 7 vloží do proměnné `subPlan`. Následně se postupně projdou potomci uzlu `coreChild`, vytvoří se pro ně plány vykonání a tyto plány se spojí s dosavadním plánem v proměnné `subPlan` pomocí algoritmu *StackTreeDesc* (řádek 19). Výsledný plán se ve dvojici s proměnnou `coreChild` vloží na řádku 22 do pole `subPlans`. Po projití všech potomků uzlu `node`, kteří jsou uzly jádra dotazu, se funkce ukončí a na výstup se vloží pole `subPlans`.

Po vytvoření plánů pro jednotlivé uzly jádra dotazu se vytvoří plán vykonání pro vstupní uzel `node` opět pomocí volání funkce `BuildPlanBinOneCost` s úpravou popsanou v předchozím odstavci. Plán se na řádku 3 vloží do proměnné `plan`. Následně se inicializuje proměnná `minCost` a `bestPlan`. `minCost` se nastaví na nejvyšší možné číslo, proměnná `bestPlan` zůstane prázdná. Poté na řádku 6 vytvoří všechny možné permutace prvků v poli `subPlans`, které se na řádcích 7 – 31 projdou. U každé permutace se projdou plány v proměnné `subPlans` v pořadí určené permutací. Tyto plány se postupně spojí s plánem v proměnné `joinPlan`. Pokud je uzel aktuálně procházené dvojice výstupní, použije se algoritmus spojení *StackTreeAnc* (řádek 14). Pokud výstupní není, použije se algoritmus spojení *StackTreeAncSrt* (řádek 20), který funguje podobně jako algoritmus *StackTreeAnc*, ale navíc řeší problém uspořádání mezivýsledku, který je popsán v kapitole 4.1. Poté se na řádku 22 spočítá cena dosavadního plánu a pokud tato cena překročila minimální nalezenou cenu, tvoření plánu se pro tuto permutaci ukončí a bude se pokračovat další permutací (řádek 24). Po projití všech plánů v proměnné `subPlans` se zkontroluje, zda je výsledný plán v proměnné `joinPlan` levnější než plán v proměnné `bestPlan` (řádek 27). Pokud ano, proměnné `minCost` a `bestPlan` se na řádcích 28 a 29 aktualizují na hodnoty `joinPlan.cost` a `joinPlan`.

Po projití všech možných permutací se v proměnné `bestPlan` nachází nejlepší možný plán pro zpracování dotazů s libovolným počtem výstupních uzlů.

4.4 Sestavení plánu s heuristickou optimalizací

Jako druhý způsob optimalizace byl zvolen heuristický přístup. U tohoto přístupu vznikne vždy pouze jeden plán, na rozdíl od předchozích algoritmů s cenovou optimalizací, který je optimalizovaný, ale nemusí být vždy nejlepší. Optimalizace opět spočívá v určení pořadí, v jakém se mají jednotlivé plány spojovat. Pořadí se určí podle velikosti výsledků plánů (přesněji řečeno podle odhadů těchto velikostí). Plány, které se mají spojit s hlavním plánem, se spustí nad vzorkem dat, zjistí se velikosti výsledků a tyto plány se seřadí podle této hodnoty vzestupně. Tím, že se budou plány spojovat

Algoritmus 13: `BuildPlanBinMultipleCost`

Vstup: node – uzel jádra dotazu z TPQ stromu
Výstup: nejlepší možný plán vykonání dotazu rozvinutý od uzlu node

```
1  Function BuildPlanBinMultipleCost(node):
2      subPlans = CreateSubPlans(node);
3      plan = BuildPlanBinOneCost(node);
4      minCost = ∞;
5      bestPlan = NULL;
6      permutations = all permutations of subPlans;
7      foreach permutation in permutations do
8          joinPlan = plan;
9          foreach tuple (coreChild, subPlan) in permutation do
10             if coreChild is output then
11                 relationship = GetRelationship(node, coreChild);
12                 ancOutput = all nodes in joinPlan;
13                 descOutput = all nodes in subPlan;
14                 joinPlan = StackTreeAnc(joinPlan, subPlan, node, coreChild, ancOutput,
                    descOutput, relationship);
15             else
16                 relationship = GetRelationship(node, coreChild);
17                 lastNode = last Query Core node on path from coreChild;
18                 ancOutput = all nodes in joinPlan;
19                 descOutput = all nodes in subPlan except lastNode;
20                 joinPlan = StackTreeAncSrt(joinPlan, subPlan, node, coreChild, ancOutput,
                    descOutput, relationship, lastNode);
21             end
22             CalculateCost(joinPlan);
23             if joinPlan.cost > minCost then
24                 vytváření plánu pro tuto permutaci může být zastaveno, pokračuje se další
                    permutací;
25             end
26         end
27         if joinPlan.cost < minCost then
28             minCost = joinPlan.cost;
29             bestPlan = joinPlan;
30         end
31     end
32     return bestPlan;
33 end
```

Algoritmus 14: CreateSubPlans

Vstup: node – uzel jádra dotazu z TPQ stromu
Výstup: pole dvojic $(uzel, plán)$, kde `uzel` je uzlem jádra dotazu a zároveň potomek uzlu node a `plán` je plán vykonání dotazu pro tento uzel

```
1  Function CreateSubPlans(node):
2      subPlans = empty array of tuples (coreChild, subPlan);
3      foreach coreChild in node.getCoreChildren() do
4          if coreChild is output then
5              subPlans.add(coreChild, BuildPlanBinMultipleCost(coreChild));
6          else
7              subPlan = BuildPlanBinOneCost(coreChild);
8              nextChild = coreChild;
9              while nextChild is not output do
10                 coreChildOfNextChild = nextChild.getCoreChild();
11                 if coreChildOfNextChild is not output then
12                     subSubPlan = BuildPlanBinOneCost(coreChildOfNextChild);
13                 else
14                     subSubPlan = BuildPlanBinMultipleCost(coreChildOfNextChild);
15                 end
16                 relationship = GetRelationship(nextChild, coreChildOfNextChild);
17                 ancOutput = coreChild;
18                 descOutput = all nodes in subSubPlan;
19                 subPlan = StackTreeDesc(subPlan, subSubPlan, nextChild,
                    coreChildOfNextChild, ancOutput, descOutput, relationship);
20                 nextChild = coreChildOfNextChild;
21             end
22             subPlans.add(coreChild, subPlan);
23         end
24     end
25     return subPlans;
26 end
```

Algoritmus 15: Upravující podmínka algoritmu `BuildPlanBinOneCost`pro použití v algoritmu `BuildPlanBinMultipleCost`

```
1 if neighbor is Query Core node then
2     subPlans.add(BuildPlanBinOneCost(neighbor));
3 end
```

od těch s nejmenším výsledkem, se dosáhne větší selektivity a spojování plánů v pozdějších krocích bude efektivnější, protože se bude zpracovávat méně XML uzlů.

Algoritmus pro zpracování dotazů se všemi výstupními uzly s heuristickou optimalizací je znázorněn v Algoritmu 16. Tento algoritmus bude nadále označován jako *BinAllHeur*. Algoritmus je podobný algoritmu s cenovou optimalizací. Nejprve se na řádku 2 vytvoří *IndexStream* operátor pro vstupní uzel a následně se na řádcích 3 – 8 vytvoří plány vykonání dotazů pro sousední uzly pomocí rekurzivního volání funkce `BuildPlanBinAllHeur`. Po vytvoření každého plánu se na řádku 6 tento plán spustí nad vzorkem dat pomocí volání funkce `estimateResultSize`. Tento krok je potřeba pro získání velikosti výsledku. Po vytvoření plánů pro všechny sousední uzly se tyto plány v proměnné `subPlans` seřadí podle velikosti výsledků od nejmenšího po největší (řádek 9). Následně se plány v tomto pořadí projdou a spojí s hlavním plánem stejným způsobem jako v algoritmu s cenovou optimalizací (řádky 10 – 22). Výsledkem je jeden konečný plán, který se nachází v proměnné `plan` a který se vloží na výstup.

Vzhledem k tomu, že úprava algoritmu s cenovou optimalizací na heuristickou optimalizaci je přímočará, další dva algoritmy pro zpracování dotazů s jedním výstupním uzlem a s více výstupními uzly nebudou popsány.

4.5 Srovnání jednotlivých algoritmů

V tabulce 4.1 jsou srovnány jednotlivé algoritmy popsané v této kapitole. Algoritmy označené hvězdičkou jsou specifické pro tuto práci. Tabulka ukazuje, které algoritmy jsou bez cenové optimalizace, které obsahují cenovou optimalizaci a u kterých je implementována heuristická optimalizace. Dále je v tabulce znázorněno, jaké druhy dotazů jsou jednotlivé algoritmy schopny zpracovat.

Algoritmus 16: `BuildPlanBinAllHeur`

Vstup: node – první uzel z TPQ stromu
Výstup: plán vykonání dotazu

```
1  Function BuildPlanBinAllHeur(node):
2      plan = IndexStream(node);
3      subPlans = empty array of tuples (neighbor, subPlan);
4      foreach neighbor in node.getNeighbors() do
5          subPlan = BuildPlanBinAllHeur(neighbor);
6          subPlan.estimateResultSize();
7          subPlans.add(neighbor, subPlan);
8      end
9      subPlans.sortByResultSize();
10     foreach tuple (neighbor, subPlan) in subPlans do
11         if neighbor is parent of node then
12             relationship = GetRelationship(neighbor, node);
13             ancOutput = all nodes in subPlan;
14             descOutput = all nodes in plan;
15             plan = StackTreeDesc(subPlan, plan, neighbor, node, ancOutput, descOutput,
                relationship);
16         else if node is parent of neighbor then
17             relationship = GetRelationship(node, neighbor);
18             ancOutput = all nodes in plan;
19             descOutput = all nodes in subPlan;
20             plan = StackTreeAnc(plan, subPlan, node, neighbor, ancOutput, descOutput,
                relationship);
21         end
22     end
23     return plan;
24 end
```

Algoritmus	Optimalizace		Výstupní uzly		
	Cenová	Heuristická	Jeden	Několik	Všechny
BinOne	✗	✗	✓	✗	✗
*BinOneCost**	✓	✗	✓	✗	✗
*BinOneHeur**	✗	✓	✓	✗	✗
BinMultiple	✗	✗	✓	✓	✓
*BinMultipleCost**	✓	✗	✓	✓	✓
*BinMultipleHeur**	✗	✓	✓	✓	✓
BinAllCost	✓	✗	✗	✗	✓
*BinAllHeur**	✗	✓	✗	✗	✓

Tabulka 4.1: Srovnání jednotlivých algoritmů

Kapitola 5

Experimenty

Testování algoritmů bylo prováděno v prototypu nativního XML databázového systému *RadegastXDB* [9]. Pro testování byly použity kolekce *TreeBank*, *XMark*, *SwissProt* a *DBLP*, které se při srovnávání výkonů algoritmů nad XML běžně používají. Pro každou kolekci bylo připraveno několik dotazů. Počty dotazů pro jednotlivé kolekce jsou uvedeny v tabulce 5.1. Celkové počty zpracovaných dotazů jsou vyšší, protože každý dotaz byl zpracováván několikrát, pokaždé s jiným počtem výstupních uzlů. Takto se počty výstupních uzlů nastavovaly postupně od jednoho až po všechny uzly. Uzly, které byly nastaveny jako výstupní, byly vybrány náhodně. Příklady dotazů pro jednotlivé kolekce jsou uvedeny v tabulce 5.3. Každý algoritmus byl spuštěn pětkrát, zaznamenaly se časy pro každý průběh a na konci zpracovávání se pro tyto hodnoty spočítal aritmetický průměr. Zprůměrování bylo provedeno tak, že se ignoroval nejlepší a nejhorší čas, aby se eliminovaly případné extrémní hodnoty způsobené např. načítáním dat do mezipaměti. Testování bylo prováděno na počítači s parametry uvedenými v tabulce 5.2.

Pro každý zpracovávaný dotaz byly zaznamenány následující údaje:

- počet výstupních uzlů,
- počet všech uzlů,
- velikost výsledku,
- časy vykonání dotazu pro jednotlivé algoritmy včetně času stráveného vytvářením plánu a optimalizací,
- časy vytvoření plánů včetně optimalizace.

Údajů bylo zaznamenáno více, ale pro potřeby popisu experimentů nejsou tyto údaje potřeba. Ze zaznamenaných údajů bylo vypočítáno zrychlení jednotlivých algoritmů oproti původnímu algoritmu založeném na binárních spojeních implementovaném v *RadegastXDB*. Zrychlení bylo vypočítáno

Kolekce	Počet připravených dotazů	Celkový počet zpracovaných dotazů
TreeBank	60	391
XMark	32	170
SwissProt	10	41
DBLP	50	158

Tabulka 5.1: Počty připravených a zpracovaných dotazů pro jednotlivé kolekce

CPU	Označení	AMD Ryzen 5 3600
	Počet jader	6
	Počet vláken	12
	Základní frekvence	3,6 GHz
	Maximální frekvence	4,2 GHz
RAM	Kapacita	16 GB
Operační systém	Windows 10	

Tabulka 5.2: Parametry PC, na kterém byly prováděny experimenty

podle vzorce $\frac{T_1}{T_2}$, kde T_1 je čas vykonání původního algoritmu a T_2 čas vykonání testovaného algoritmu. Z těchto hodnot byl vytvořen krabicový graf, ve kterém bylo na osu y aplikováno logaritmické měřítko. V takovém grafu hodnota 1 znamená, že čas vykonání testovaného algoritmu byl stejný jako čas vykonání původního algoritmu, hodnoty vyšší než 1 značí, že testovaný algoritmus byl rychlejší než původní algoritmus a hodnoty menší než 1 značí, že testovaný algoritmus byl pomalejší.

5.1 Stanovení cenového modelu

Aby bylo možné stanovit cenu plánu vykonání, je potřeba stanovit cenový model. Tento model byl vytvořen následujícím postupem. Nad kolekcemi XMark, SwissProt a DBLP bylo spuštěno 10 dotazů, které prováděly spojení dvou uzlů. Pro každý dotaz se testovaly všechny algoritmy binárních

Kolekce	Příklad dotazu
TreeBank	`//EMPTY/.S[./_PERIOD_ and ./NP//NN and .//ADJP/PP[.//IN and .//PRP_DOLLAR_]]`
XMark	`//closed_auctions//annotation[.//happiness and ./description/text]`
SwissProt	`//Entry[./Ref//Comment and .//Descr]`
DBLP	`//article[.//title[./sub and .//i] and ./year and ./number]`

Tabulka 5.3: Příklady dotazů pro jednotlivé kolekce

spojení a u každého algoritmu se zvlášť rozlišovaly vztahy AD a PC. Lineární regresí se stanovily čtyři konstanty tak, aby se odhadnutý čas co nejvíce blížil skutečnému času. Tyto konstanty jsou použity při odhadu ceny jednotlivých operátorů. Cena operátoru je stanovena následujícím vztahem: $T = A \cdot sizeof(input_{left}) + B \cdot sizeof(input_{right}) + C \cdot sizeof(output) + D$, kde A, B, C a D jsou jednotlivé konstanty získané lineární regresí, $input_{left}$ je levý vstup operátoru, $input_{right}$ je pravý vstup operátoru a *output* je výstup operátoru. Funkce *sizeof* vrací velikost výsledku operátoru. Pomocí tohoto vztahu je možno pro každý operátor vypočítat jeho cenu. Sečtením cen všech operátorů v plánu vykonání dotazu získáme cenu celého plánu.

Jak je vidět z popsaného vztahu, pro určení ceny operátoru je nutné znát velikosti vstupů a také velikost výstupu. Operátor je tedy nutné nejprve spustit, abychom tyto informace získali. Kdyby se ale operátor spouštěl nad celou databází, která může být velice rozsáhlá, bylo by stanovení ceny značně neefektivní. Z tohoto důvodu se cena operátoru ve všech algoritmech v této práci pouze odhaduje na základě menšího vzorku databáze. Tento vzorek byl vytvářen s faktorem 1/200 a je tedy přibližně 200x menší než původní databáze. Spuštění operátoru nad vzorkem a odhad jeho ceny je tedy v porovnání s časem vykonání dotazu nad celou databází velice rychlý. U tohoto přístupu může ale nastat problém, pokud by vzorek dat nereflektoval distribuci dat v celé databázi. V takovém případě by odhad nebyl přesný a plán vykonání dotazu by nebyl optimalizovaný.

5.2 Materializace

Pro účely testování byl vytvořen speciální operátor *Materialize*, který umožňuje materializaci výsledku vstupního operátoru a může být uplatněn při odhadování velikosti výsledku nad vzorkem. Materializace výsledku spočívá v tom, že výsledek vstupního operátoru se uloží do paměti operátoru *Materialize* a při dalším spuštění plánu se nebudou tyto výsledky znovu získávat ze vstupního operátoru, ale použijí se ty, které jsou uložené v paměti. Tímto se ušetří výpočetní výkon při sestavování plánu, kdy se v určitých případech operátory spouští několikrát. Tato situace nastává tehdy, když se vytvoří plán P pro nějaký uzel. Tento plán se poté spojí s jiným plánem Q a výsledný plán se poté spojí zase s dalším plánem R. Po každém spojení je potřeba spočítat cenu dosavadního plánu. Jak již bylo zmíněno v kapitole 5.1, pro výpočet ceny plánu je nutné znát velikosti vstupních operátorů a velikost výstupu. Pro zjištění těchto informací je tedy nutné plán spustit nad vzorkem dat. Po vytvoření plánu P se tedy tento plán spustí. Po vytvoření plánu Q se tento plán také spustí, tudíž se spustí i plán P. Pro výpočet ceny ceny plánu R je potřeba další spuštění jednotlivých podplánů. V tomto případě musel být plán P vyhodnocován již třikrát, přitom vždy vracel stejný výsledek. Tento problém řeší právě operátor *Materialize*, který zajistí, že plán, který již byl spuštěn, nemusí být spuštěn znovu a využije se již vyhodnocený výsledek z předchozího spuštění.

Tento operátor se používá pouze ve fázi sestavování plánu, kdy se uvažované plány opakovaně spouští pouze nad vzorkem databáze. Před samotným spuštěním dotazu nad úplnou databází musí

být tento operátor z celého plánu vykonání odstraněn. V opačném případě by způsobil velice neefektivní vykonávání dotazu.

5.3 Srovnání jednotlivých algoritmů

V této části budou srovnány jednotlivé algoritmy pro tři kategorie dotazů.

5.3.1 Dotazy s jedním výstupním uzlem

V této kategorii experimentů byly zpracovávány dotazy s jedním výstupním uzlem. Na těchto dotazech byly testovány následující algoritmy: *BinOneCost*, *BinOneCostMater*, *BinOneHeur*, *BinMultipleCost*, *BinMultipleCostMater* a *BinMultipleHeur*.

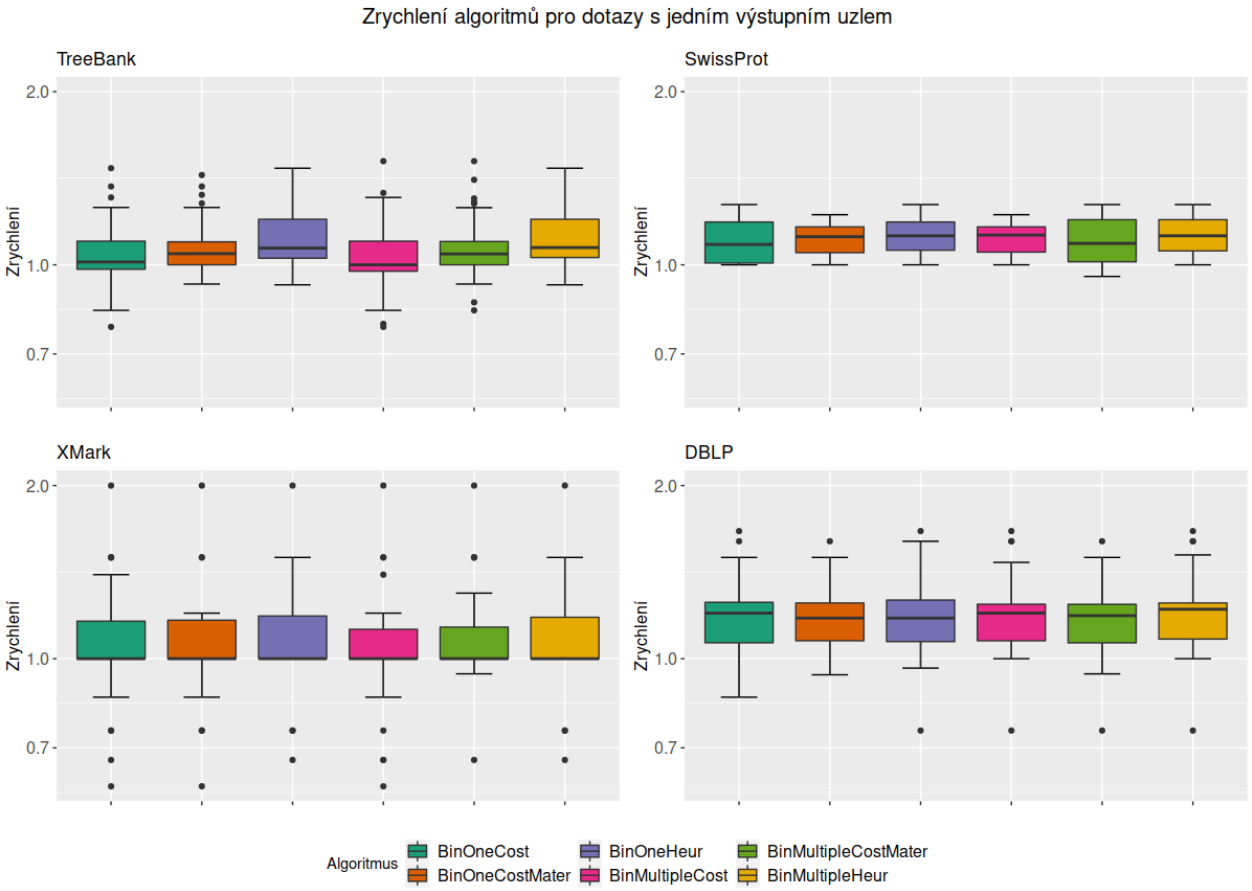
Krabicové grafy zrychlení zpracování dotazů pro jednotlivé kolekce jsou na obrázku 5.1. Na všech čtyřech grafech je vidět, že všechny algoritmy byly průměrně rychlejší oproti původnímu algoritmu. Algoritmy mezi sebou dosahovaly přibližně stejného zrychlení. Menší rozdíl nastal u kolekce TreeBank, kde bylo zrychlení heuristicky optimalizovaných algoritmů *BinOneHeur* a *BinMultipleHeur* o pár bodů vyšší. Rozptyl hodnot zrychlení byl mezi kolekcemi srovnatelný, pouze u kolekce XMark se u některých dotazů dosáhlo zrychlení o hodnotě 2, na druhou stranu u jiných dotazů došlo naopak ke značnému zpomalení, kdy se hodnoty pohybovaly okolo 0,6.

Na obrázku 5.2 se nachází grafy poměrů celkových časů zpracování dotazu k časům sestavení plánu. Jak je na grafech vidět, čas sestavení plánu pro všechny dotazy a kolekce je v poměru k celkovému času vykonání zanedbatelný. Tento čas je nejvyšší u algoritmů *BinOneCost* a *BinMultipleCost*, ale stále se jedná pouze o jednotky procent v poměru k celkovému času vykonání dotazu. Vyšší čas strávený sestavováním plánů je dán optimalizací, kdy se v těchto algoritmech sestavuje velké množství plánů, ze kterých se následně vybere ten nejlevnější. Dále je na grafech možno vidět, že použití materializace u algoritmů *BinOneCostMater* a *BinMultipleCostMater* výrazně snížilo čas sestavování plánů a např. u kolekce TreeBank se tato úspora projevila v celkovém času vykonání dotazu (obrázek 5.1). Nejlépe na tom jsou algoritmy *BinOneHeur* a *BinMultipleHeur*, u kterých byly časy sestavení plánu velice nízké a občas i neměřitelné.

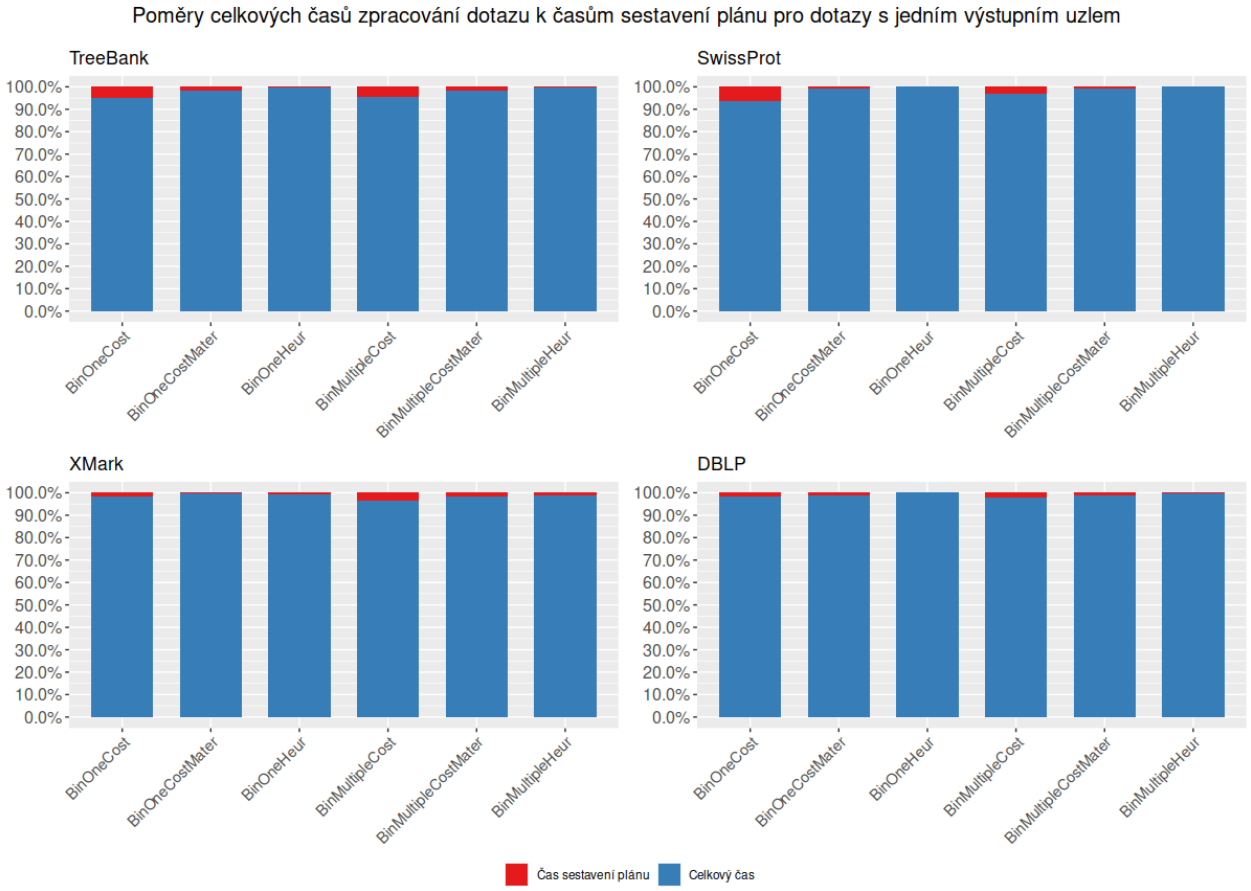
5.3.2 Dotazy s více výstupními uzly

V této kategorii experimentů byly zpracovávány dotazy s více výstupními uzly. Na těchto dotazech byly testovány následující algoritmy: *BinMultipleCost*, *BinMultipleCostMater* a *BinMultipleHeur*.

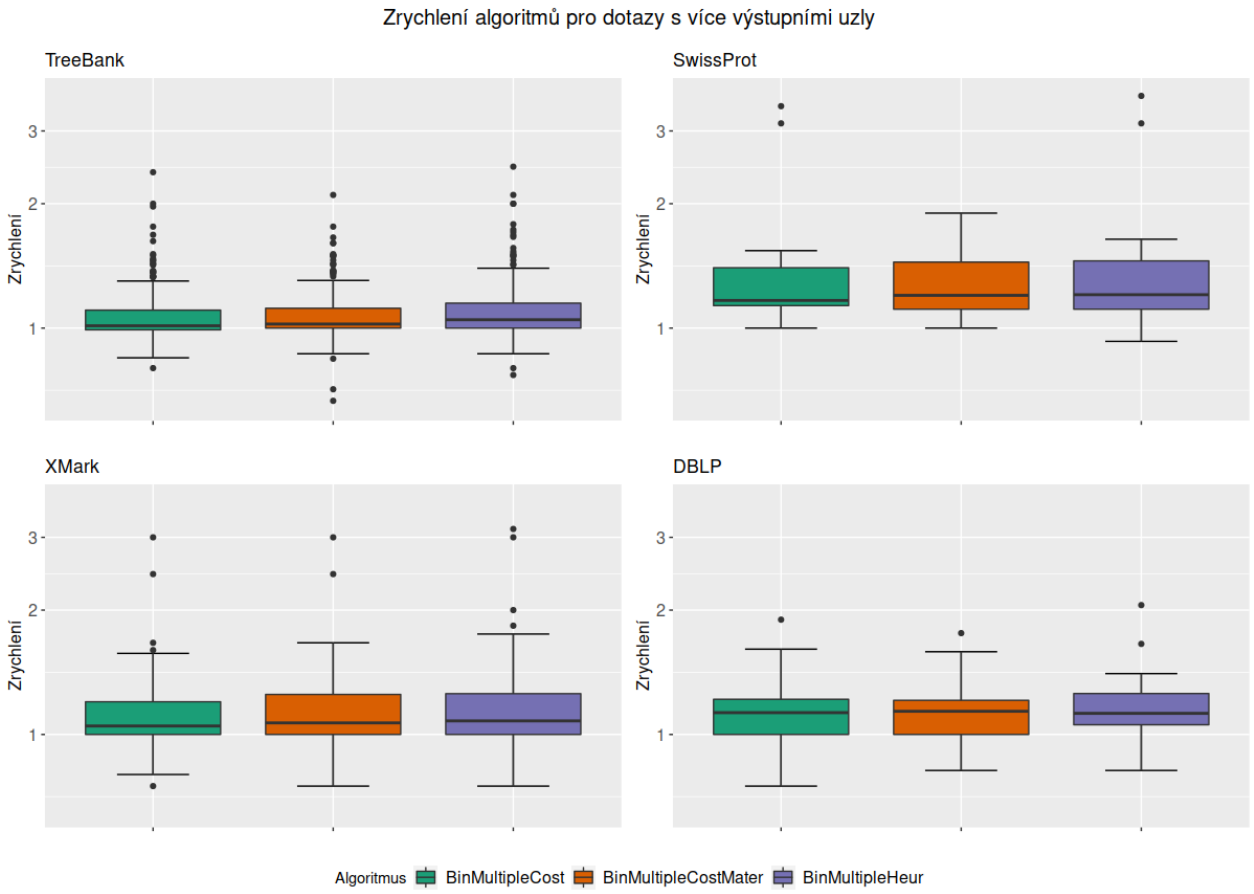
Krabicové grafy zrychlení zpracování dotazů pro jednotlivé kolekce se nachází na obrázku 5.3. Algoritmy opět dosahovaly u většiny dotazů zrychlení oproti původnímu algoritmu. Největšího průměrného zrychlení se v tomto případě dosáhlo u kolekce SwissProt, u které bylo navíc u dvou dotazů dosaženo zrychlení vyšší než 3. Tyto dotazy budou popsány později v kapitole 5.3.4.



Obrázek 5.1: Krabicový graf zrychlení zpracování dotazů s jedním výstupním uzlem



Obrázek 5.2: Poměry celkových časů zpracování dotazu k časům sestavení plánu pro dotazy s jedním výstupním uzlem



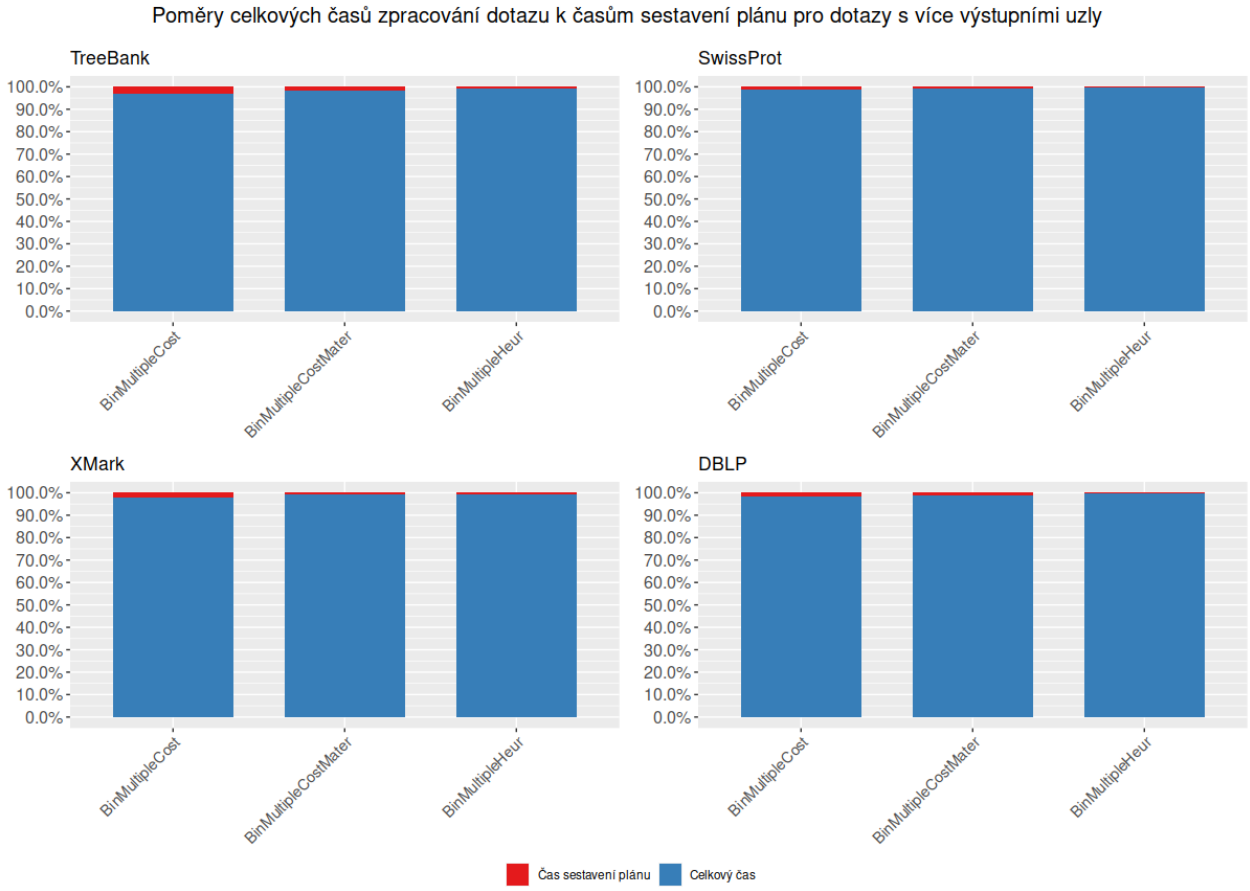
Obrázek 5.3: Krabicový graf zrychlení zpracování dotazů s více výstupními uzly

Stejně jako u zpracování dotazů s jedním výstupním uzlem je i zde čas sestavení plánu v poměru k celkovému času vykonání u všech kolekcí zanedbatelný. Grafy těchto poměrů se nachází na obrázku 5.4. Opět platí, že nejvíce času sestavováním plánu strávil algoritmus *BinMultipleCost* a nejlépe na tom je algoritmus *BinMultipleHeur*.

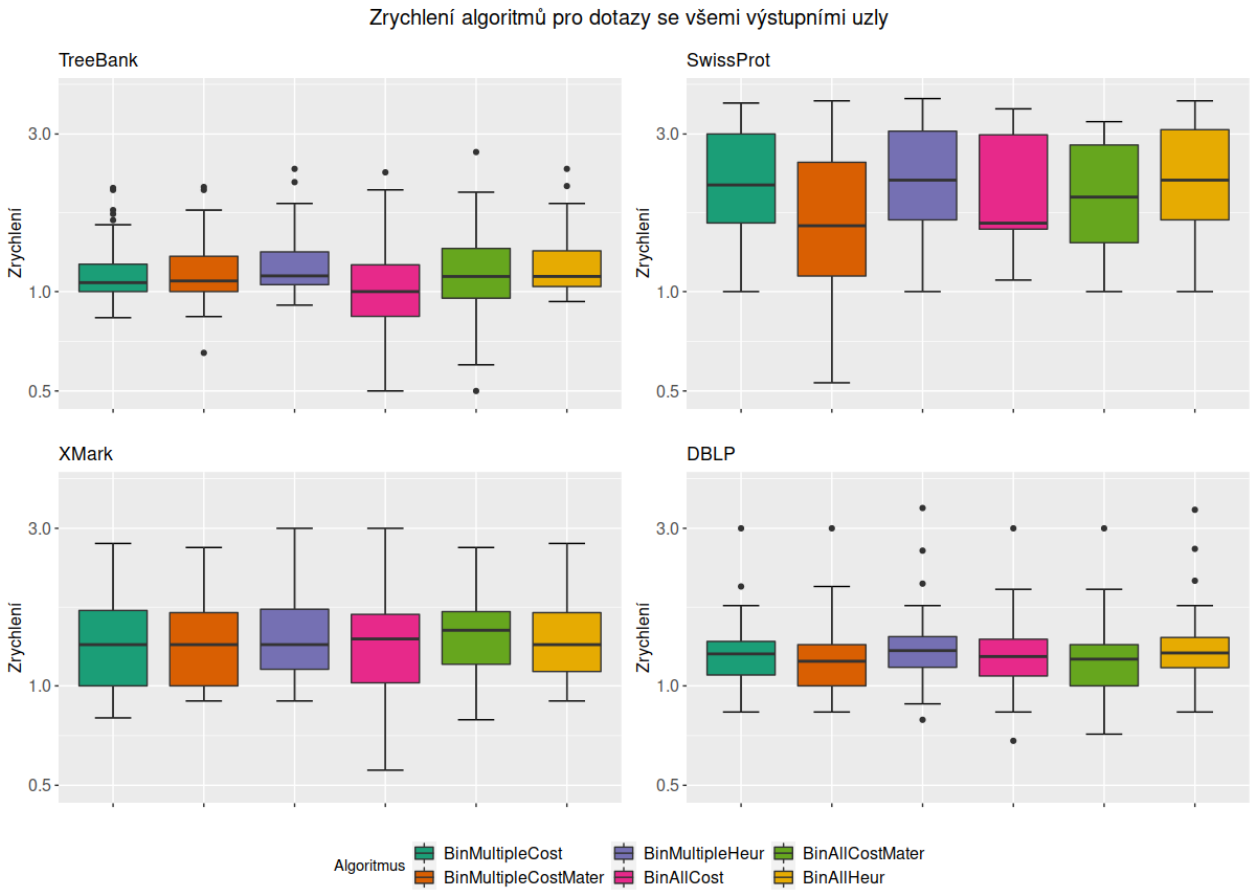
5.3.3 Dotazy se všemi výstupními uzly

V této kategorii experimentů byly zpracovávány dotazy se všemi výstupními uzly. Na těchto dotazech byly testovány následující algoritmy: *BinMultipleCost*, *BinMultipleCostMater*, *BinMultipleHeur*, *BinAllCost*, *BinAllCostMater*a *BinAllHeur*.

Krabicové grafy zrychlení zpracování dotazů pro jednotlivé kolekce se nachází na obrázku 5.5. V této kategorii dotazů byly všechny algoritmy kromě algoritmu *BinAllCost* u kolekce TreeBank rychlejší než původní algoritmus. Algoritmus *BinAllCost* dosahoval u kolekce TreeBank jak zrychlení, tak zpomalení a průměrná hodnota zrychlení se ustálila na hodnotě 1, což znamená, že průměrně ke zrychlení nedošlo. Naopak u kolekce SwissProt došlo k výraznému zrychlení, kdy se prů-



Obrázek 5.4: Poměry celkových časů zpracování dotazu k časům sestavení plánu pro dotazy s více výstupními uzly



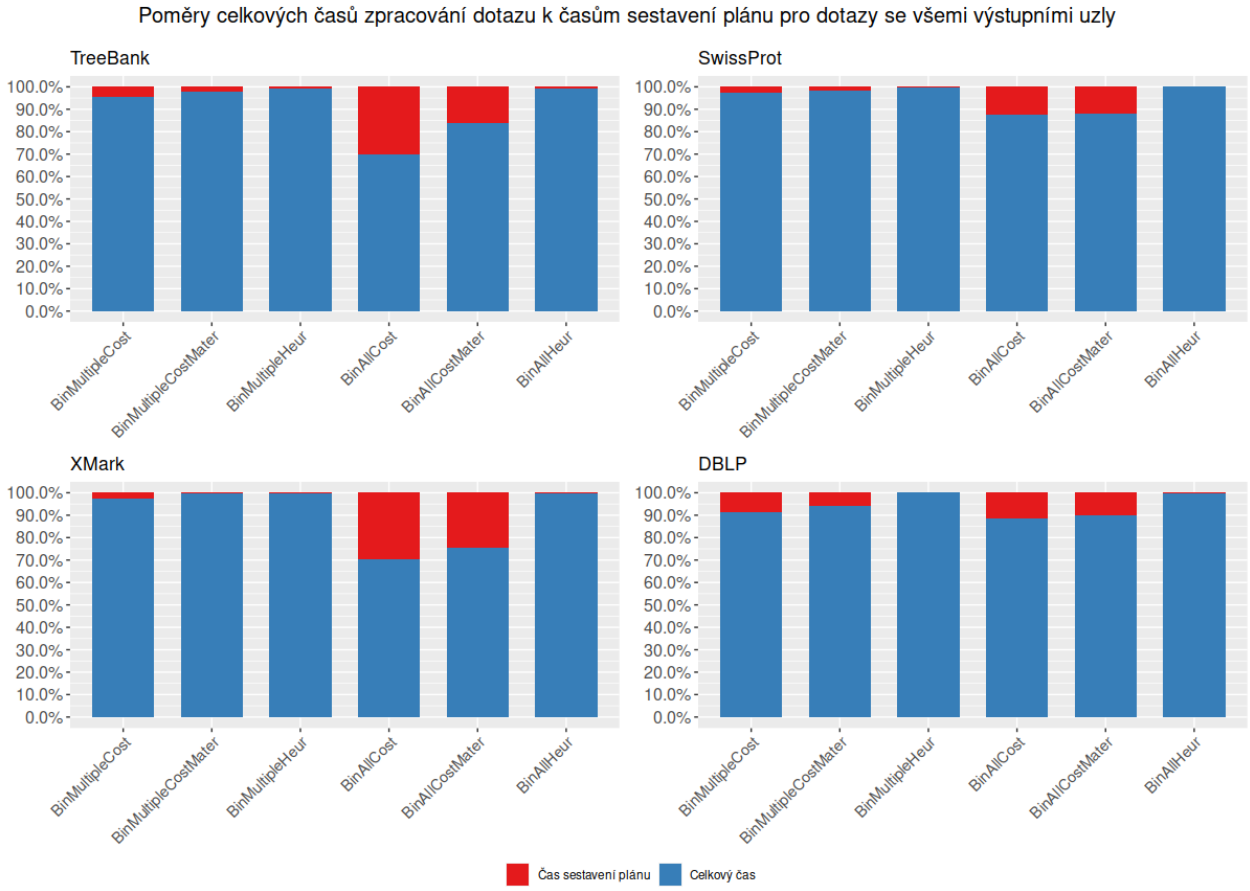
Obrázek 5.5: Krabicový graf zrychlení zpracování dotazů se všemi výstupními uzly

měrné hodnoty zrychlení u všech algoritmů pohybovaly mezi hodnotou 2 a 3. U kolekce DBLP došlo ke zrychlení okolo hodnoty 3 také, ale pouze u jednoho dotazu.

Jak je vidět na obrázku 5.6, pro kolekci TreeBank celkový čas sestavování plánů u algoritmu *BinAllCost* dosahuje 30 % z celkového času vykonání plánů a u algoritmu *BinAllCostMater* se jedná o 16 %. U kolekce XMark jsou tyto hodnoty přibližně stejné, a to 30 % a 25 %. U dalších kolekcí jsou hodnoty u těchto algoritmů menší, ale oproti jiným algoritmům stále vyšší, a to kolem 12 %. U ostatních algoritmů se jedná o hodnoty do 10 % pro všechny kolekce. Jde vidět, že se vysoké časy u kolekcí TreeBank a XMark projevily i na celkových časech vykonání dotazu (obrázek 5.5). Opět platí, že časy sestavování plánů u heuristicky optimalizovaných algoritmů jsou u všech kolekcí zanedbatelné.

5.3.4 Extrémní dotazy

U některých dotazů došlo u určitých algoritmů k extrémním hodnotám zrychlení. Dotazy, u kterých se optimalizace vyplatila a došlo k velkému zrychlení, jsou následující:



Obrázek 5.6: Poměry celkových časů zpracování dotazu k časům sestavení plánu pro dotazy se všemi výstupními uzly

- `//Entry[.//Descr and .//Ref[./Cite and ./Comment]]` – SwissProt, zrychlení 3,44
- `//open_auctions//open_auction[./current and .//bidder and ./initial]` – XMark, zrychlení 3,14
- `//inproceedings[.//author and ./title and ./url and .//crossref]` – DBLP, zrychlení 3,46
- `//EMPTY[.//NP and .//PP//NN]` – TreeBank, zrychlení 2,45

U některých dotazů došlo naopak k výraznému zpomalení. Tyto dotazy jsou následující:

- `//categories/category[.//name and ./description/text]` – XMark, zrychlení 0,6
- `//EMPTY//SINV[./_COMMA_ and .//_PERIOD_]` – TreeBank, zrychlení 0,66

5.3.5 Shrnutí experimentů

V tabulce 5.4 jsou seřazeny algoritmy dle jejich efektivity určené na základě experimentů. Nejlepších výsledků dosahovaly algoritmy s heuristickou optimalizací. U těchto algoritmů se plán vykonání tvoří tak, aby v každém kroku vykonávání došlo k co nejvyšší možné selektivitě a tudíž menšímu množství zpracovávaných dat. Výhodou tohoto algoritmu je také čas optimalizace, který je ze všech algoritmů nejnižší – poměry časů sestavování plánu se pohybují pod jedním procentem z celkové doby vykonávání dotazu.

Na druhém místě jsou algoritmy s cenovou optimalizací s využitím materializace. Plány získané těmito algoritmy jsou stejné jako plány získané algoritmy bez materializace, ale právě díky použití materializace je proces optimalizace rychlejší, tudíž z pohledu celkového času vykonání dotazu jsou tyto algoritmy efektivnější.

Na posledním místě jsou algoritmy s cenovou optimalizací bez použití materializace. Tyto algoritmy jsou z implementovaných algoritmů nejméně efektivní, ale stále jsou efektivnější než původní algoritmy bez optimalizace. Nutno podotknout, že pro některé dotazy byly optimalizované algoritmy pomalejší než algoritmy bez optimalizace. Záleží tedy na různých okolnostech, kdy je daný optimalizovaný algoritmus výhodnější. Obecně ale platí, že se optimalizované algoritmy vyplatí použít.

Umístění	**Algoritmus**
1	*BinOneHeur* *BinMultipleHeur* *BinAllHeur*
2	*BinOneCostMater* *BinMultipleCostMater* *BinAllCostMater*
3	*BinOneCost* *BinMultipleCost* *BinAllCost*

Tabulka 5.4: Srovnání efektivity jednotlivých algoritmů

Kapitola 6

Závěr

Cílem této diplomové práce byla implementace vybraných algoritmů pro sestavení optimalizovaného plánu dotazu za použití statistik XML kolekce. Tento cíl byl úspěšně splněn.

Celkově bylo implementováno 8 algoritmů pro sestavení plánu vykonání dotazu, z toho 6 algoritmů vytváří optimalizovaný plán a 5 z těchto algoritmů je specifických pro tuto diplomovou práci. Pomocí testování algoritmů s experimentálními kolekcemi a TPQ dotazy bylo dokázáno, že optimalizované algoritmy vytváří efektivnější plán vykonání dotazu. Bylo také ukázáno, že poměr času sestavení a optimalizace plánu k času celkového vyhodnocení dotazu je ve většině případů zanedbatelný.

Pomocí experimentů bylo ukázáno, že nejefektivnějšími algoritmy jsou algoritmy s heuristickou optimalizací, poté algoritmy s cenovou optimalizací s materializací a na posledním místě algoritmy s cenovou optimalizací bez materializace.

Literatura

1. W3C. *Extensible Markup Language (XML) 1.0*. 1998-02-10. Dostupné také z: `https://www.w3.org/TR/1998/REC-xml-19980210`.
2. W3C. *XML Path Language (XPath) Version 1.0*. 1999-11-16. Dostupné také z: `https://www.w3.org/TR/1999/REC-xpath-19991116`.
3. W3C. *XQuery 1.0: An XML Query Language*. Dostupné také z: `https://www.w3.org/TR/2007/REC-xquery-20070123`.
4. ZHANG, Chun; NAUGHTON, Jerey; DEWITT, David; LUO, Qiong; LOHMAN, Guy. On Supporting Containment Queries in Relational Database Management Systems. *ACM SIGMOD Record*. 2001-03. ISBN 1581133324.
5. LUKÁŠ, Petr. *Structural XML Query Processing*. 2019. Dostupné také z: `http://hdl.handle.net/10084/139029`. Vysoká škola báňská - Technická univerzita Ostrava.
6. AL-KHALIFA, S.; JAGADISH, H. V.; KOUDAS, N.; PATEL, J. M.; SRIVASTAVA, D.; YU-QING WU. Structural joins: a primitive for efficient XML query pattern matching. In: *Proceedings 18th International Conference on Data Engineering*. 2002, s. 141–152. Dostupné z DOI: `10.1109/ICDE.2002.994704`.
7. WU, Y.; PATEL, J.M.; JAGADISH, H.V. Structural join order selection for XML query optimization. In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 2003, s. 443–454. Dostupné z DOI: `10.1109/ICDE.2003.1260812`.
8. LUKÁŠ, Petr; BAČA, Radim; KRÁTKÝ, Michal; LING, Tok Wang. Demythization of Structural XML Query Processing: Comparison of Holistic and Binary Approaches. *IEEE Transactions on Knowledge and Data Engineering*. 2019, roč. 33, č. 4, s. 1439–1452. Dostupné z DOI: `10.1109/TKDE.2019.2946157`.
9. LUKÁŠ, Petr; BAČA, Radim; KRÁTKÝ, Michal. RadegastXDB - Prototype of Native XML Database Management System: Technical Report. *CoRR*. 2019, roč. abs/1903.03761. Dostupné také z: `http://arxiv.org/abs/1903.03761`.

Příloha A

Testovací data

Součástí přílohy k této práci jsou následující soubory, které se týkají experimentů:

- Soubor `experiments_data.xlsx` obsahující data z experimentů. Jedná se o Excel soubor, ve kterém jsou data z experimentů rozdělena do jednotlivých listů dle kolekcí.
- Adresář `TPQs` obsahující soubory s testovacími TPQ dotazy pro jednotlivé kolekce.

Příloha B

Zdrojové soubory

Součástí přílohy k práci jsou zdrojové soubory obsahující implementované algoritmy. Zdrojové soubory jsou umístěny v adresáři `src/Xdb2`. Jedná se o projekt v programu Microsoft Visual Studio a cesta k samotnému projektovému souboru je následující: `test/xdb2/Xdb2.sln`. Jedná se o kompletní databázi, přičemž změny v rámci této práce byly prováděny v několika souborech. V adresáři `test/xdb2/QueryTesting` byly upraveny nebo vytvořeny následující soubory:

- `main.cpp` – hlavní soubor, ze kterého se spouští experimenty
- `bjTests.cpp` – soubor obsahující funkci `testTpqQueries()`, která provede spuštění samotných experimentů, tato funkce se volá z metody `main()` v souboru `main.cpp`
- `bjTests.h` – hlavičkový soubor pro `bjTests.cpp`
- `bjAlgorithms.cpp` – soubor obsahující všechny algoritmy implementované v rámci této práce, obsahuje také metodu `testPlan()`, která otestuje určitý algoritmus pro daný TPQ dotaz, algoritmus i TPQ dotaz jsou funkci předány pomocí parametru
- `bjAlgorithms.h` – hlavičkový soubor pro `bjAlgorithms.cpp`

V adresáři `xdb2/algebra/algorithms/ttp_binary` byly upraveny následující soubory:

- `cIndexNodeStreamB.cpp`
- `cMaterialize.cpp`
- `cMicroOperator.h`
- `cStackFilterAncAD.cpp` a `cStackFilterAncAD.h`
- `cStackFilterAncPC.cpp` a `cStackFilterAncPC.h`
- `cStackFilterDescAD.cpp` a `cStackFilterDescAD.h`

- `cStackFilterDescPC.cpp` a `cStackFilterDescPC.h`
- `cStackTreeAnc.cpp` a `cStackTreeAnc.h`
- `cStackTreeDesc.cpp` a `cStackTreeDesc.h`

V těchto existujících souborech byla doplněna funkcionalita umožňující vytváření optimalizovaných dotazů (výpočet a uložení ceny operátoru).

Pro zprovoznění databáze je potřeba vnitřní projekty zkompilovat v tomto pořadí:

- QuickDB
- Xdb
- QueryTesting